Contribution à la mise-en-œuvre d'un moteur d'exécution de modèles UML pour la simulation d'applications temporisées et concurrentes

Benyahia Abderraouf

N° d'ordre : 2012-23-TH

THÈSE DE DOCTORAT

DOMAINE : STIC
Spécialité : Informatique

Ecole Doctorale « Sciences et Technologies de l'Information des Télécommunications et des Systèmes »

Présentée par :

Abderraouf BENYAHIA

Contribution à la mise-en-œuvre d'un moteur d'exécution de modèles UML pour la simulation d'applications temporisées et concurrentes

Soutenue le 26 novembre 2012 devant les membres du jury :

Mme. Annie CHOQUET-GENIET	Université de Poitiers, LIAS	Rapporteur
M. Laurent PAUTET	Telecom ParisTech	Rapporteur
M. François TERRIER	CEA LIST	Directeur de thèse
M. Arnaud CUCURRU	CEA LIST	Co-encadrant
M. Jean-Paul SANSONNET	LIMSI-CNRS	Examinateur
M. Julien DEANTONI	Université de Nice Sophia Antipolis	Examinateur
M. Laurent RIOUX	THALES TRT	Examinateur
Mme. Yolaine BOURDA	Supélec	Examinateur
M. Frédéric BOULANGER	Supélec	Invité
M. Safouan TAHA	Supélec	Invité

Résumé

Ce travail s'inscrit dans le cadre de L'Ingénierie Dirigée par les Modèles (IDM) et la sémantique d'exécution du langage UML appliqué à l'exécution de modèles des applications temps réel embarquées. Dans ce contexte, l'OMG propose une norme qui définit un modèle d'exécution pour un sous-ensemble d'UML appelé fUML (foundational UML subset). Ce modèle d'exécution définit une sémantique précise non ambigüe facilitant la transformation de modèles, l'analyse, l'exécution de modèles et la génération de code.

L'objectif de cette thèse est d'étudier et mettre-en-œuvre un moteur d'exécution de modèles UML pour les systèmes temps réel embarqués en explicitant les hypothèses portant sur la sémantique d'exécution des modèles à un niveau d'abstraction élevé afin de mettre en œuvre une possibilité d'exécution le plus tôt possible dans le flot de conception de l'application. Pour cela, nous avons étendu le modèle d'exécution défini dans fUML, en apportant une contribution sur trois aspects importants concernant les systèmes temps réel embarqués :

- **Gestion de la concurrence :** fUML ne fournit aucun mécanisme pour gérer la concurrence dans son moteur d'exécution. Nous répondons à ce problème par l'introduction d'un ordonnanceur explicite permettant de contrôler les différentes exécutions parallèles, tout en fournissant la flexibilité nécessaire pour capturer et simuler différentes politiques d'ordonnancements.
- **Gestion du temps :** fUML ne fixe aucune hypothèse sur la manière dont les informations sur le temps sont capturées ainsi que sur les mécanismes qui les traitent dans le moteur d'exécution. Pour cela, nous introduisons une horloge, en se basant sur le modèle de temps discret, afin de prendre en compte les contraintes temporelles dans les exécutions des modèles.
- **Gestion des profils :** les profils ne sont pas pris en compte par ce standard, cela limite considérablement la personnalisation du moteur d'exécution pour prendre en charge de nouvelles variantes sémantiques. Pour répondre à ce problème, nous ajoutons les mécanismes nécessaires qui permettent l'application des profils et la capture des extensions sémantiques impliquées par l'utilisation d'un profil.

Une implémentation de ces différentes extensions est réalisée sous forme d'un plugin Eclipse dans l'outil de modélisation Papyrus UML.

Table des matières

CHAPITRE 1

Introduction

L'Ingénierie Dirigée par les Modèles (IDM) place les modèles au cœur des processus d'ingénierie logicielle et système. L'hypothèse est que différents modèles d'un même système, correspondant à différentes vues et/ou niveaux d'abstraction, peuvent cohabiter au sein d'un processus de développement. Chaque modèle fournit une abstraction adaptée à un type d'exploitation particulier tel que l'analyse d'ordonnancement, la validation, la génération de code, etc., l'ensemble permettant idéalement de maitriser la complexité des logiciels et d'améliorer la rapidité et la qualité des processus de développement.

Le Model Driven Architecture (MDA) est une initiative de l'Object Management Group (OMG) définissant un cadre conceptuel, méthodologique et technologique pour la mise-en-œuvre de flots de conception basés sur l'IDM. Le cadre conceptuel identifie la nature des modèles impliquées (modèles de plateforme, modèles indépendants ou spécifiques à une plateforme) et des relations qui les unissent (abstraction, raffinement, etc.). Le cadre méthodologique préconise une approche générative par transformations de modèles pour le raffinement des différents modèles impliqués, jusqu'à la production d'un code exécutable. Le cadre technologique s'appuie sur une utilisation des formalismes normalisés par l'OMG pour la mise-en-œuvre du flot (UML pour la modélisation, QVT pour les transformations, etc.).

Confrontée à une complexité croissante des systèmes à concevoir (toujours plus de fonctionnalités à intégrer face à des contraintes impondérables de temps de mise sur le marché), l'industrie des systèmes temps-réels embarqués a naturellement identifié dans l'IDM, et dans le MDA en particulier, une opportunité pour rationaliser les flots de conception. D'une part, la structuration des flots en niveaux d'abstractions et la mise-en-œuvre d'approches génératives permettent d'éviter un certain nombre d'écueils, en automatisant (au moins partiellement) des tâches de raffinement systématiques et facilement sujettes à erreurs lorsqu'elles sont accomplies manuellement. D'autre part, le fait de

s'appuyer sur un cadre technologique commun facilite (même s'il ne suffit pas pour le garantir) une forme d'interopérabilité entre les différents acteurs et domaines de compétence du flot.

Dans les deux cas, les qualités intrinsèques des formalismes préconisés par le MDA, notamment en termes d'expressivité et d'extensibilité (cf. UML et ses profils ch. 2), favorisent la mise-en-œuvre des flots en permettant la prise en compte des particularités liées au domaine ou à l'acteur. En pratique, cette vision s'est déjà concrétisée sous la forme d'investissements significatifs de la part d'acteurs majeurs du domaine, ayant débouchés sur des réalisations concrètes tant au niveau des outilleurs [1] que des grands systémiers (Airbus, Renault, etc.). Si ces réalisations peuvent être considérées comme des avancées significatives pour le domaine (SCADE System lauréat du prix Best of show "Embeddy" à Embedded World 2011 [1][2]), un certain nombre de verrous scientifiques et technologiques persistent.

Dans le domaine des systèmes temps-réel embarqués, que l'on vise des systèmes à prépondérance logicielle ou matérielle, on cherche à produire des systèmes exécutables au sens informatique du terme. Dans les flots traditionnels basés sur le MDA, les premières formes exécutables du système sont généralement obtenues par génération de code, après une ou plusieurs étapes de raffinement manuel et/ou automatique par transformation de modèles. Ces formes exécutables sont utilisées pour mettre en œuvre des phases d'expérimentation avant obtention du système final. En d'autres termes, le comportement attendu du système final est simulé au niveau d'abstraction capturé par la forme exécutable. Pour les systèmes temps-réels embarqués, par nature contraints en ressources et en temps, la simulation est utilisée pour tester le comportement du système non seulement d'un point de vue fonctionnel, mais aussi d'un point de vue temporel et concurrentiel.

Cet état de fait signifie que, dès les niveaux d'abstraction les plus élevés du flot (donc avant raffinement), des hypothèses sur la sémantique d'exécution des différents modèles existent nécessairement, dans le sens où elles orientent la définition des règles de raffinement menant à l'obtention du code exécutable. En pratique, on constate que :

a. ces hypothèses peuvent être sous-exploitées. En effet, le niveau de détail requis pour obtenir une représentation exécutable par génération de code implique que les phases de simulation arrivent relativement tard dans le flot de conception. En exploitant effectivement les hypothèses liées à la sémantique d'exécution des différents modèles intermédiaires, la simulation pourrait intervenir plus tôt dans le flot, et pourrait donc contribuer à améliorer les flots de conception.

b. si ces hypothèses sont effectivement exploitées, leur exploitation est sujette à caution. Une pratique usuelle consiste en effet à transformer les modèles vers différents formalismes, chaque formalisme cible bénéficiant d'une sémantique bien définie, et étant adapté à un type d'exploitation particulière. Le problème est que, quelle que soit la rigueur et le soin apporté à la mise-en-œuvre des transformations,

elles partent toujours d'une spécification en langage naturel (pratique usuelle pour décrire les aspects sémantiques dans les spécifications OMG), nécessairement sujette à interprétation. En d'autres termes, la projection ne peut être considérée comme valide que par rapport à l'interprétation du formalisme en entrée. La critique s'applique également lorsque la transformation concerne le raffinement d'un modèle au sein même du flot de conception.

L'objectif principal de cette thèse est de contribuer à l'étude et la mise-en-œuvre d'un moteur d'exécution de modèles répondant au point **a.** ci-dessus, tout en anticipant le point **b.** Pour cela, la proposition que nous allons développer doit répondre à quatre critères essentiels :

1. Impliquer un effort limité quant à l'intégration dans un flot de conception respectant les préconisations du MDA, et donc une certaine proximité avec les formalismes utilisés pour mettre en œuvre ce type de flots,
2. Autoriser une certaine flexibilité pour prendre en compte les particularités de différents domaines, notamment en liens avec l'utilisation de profils UML,
3. S'appuyer sur une fondation sémantique précise et standardisée, non seulement pour expliciter les hypothèses relatives à la sémantique d'exécution supportée par le moteur, mais aussi pour pouvoir, à terme, rendre ces hypothèses disponibles pour toute projection sémantique requise par une activité d'analyse,
4. Permettre de simuler le comportement d'applications temporisées et concurrentes, en lien avec les besoins du temps-réel embarqué, et ce à des niveaux d'abstractions relativement élevés.

Nous démontrons que le le moteur d'exécution défini dans le standard OMG « Semantics of a Foundational Subset for Executable UML Models », qui capture la sémantique d'exécution d'un sous-ensemble de UML nommé fUML (foundational UML), est une base nécessaire pour atteindre ces objectifs. Nous montrons comment cette base peut devenir suffisante pour une utilisation dans le domaine des systèmes temps-réels embarqués, en apportant une contribution sur les axes suivants :

- **Gestion de la concurrence :** fUML ne fournit aucun mécanisme pour gérer la concurrence dans son moteur d'exécution. Nous répondons à ce problème par l'introduction d'un ordonnanceur explicite permettant de contrôler les différentes exécutions parallèles, tout en fournissant la flexibilité nécessaire pour capturer et simuler différentes politiques d'ordonnancements.

- **Gestion du temps :** fUML ne fixe aucune hypothèse sur la manière dont les informations sur le temps sont capturées ainsi que sur les mécanismes qui les traitent dans le moteur d'exécution. Pour cela, nous introduisons une horloge, en se basant sur le modèle de temps discret, afin de prendre en compte les contraintes temporelles dans

les exécutions des modèles. Cette horloge permet d'une part l'étiquetage temporel des exécutions, c'est-à-dire la construction des pas de temps des exécutions. D'autre part, elle permet de déclencher des exécutions contraintes par le temps, par exemple dans le cas de tâches périodiques.

- **Gestion des profils, pour permettre une certaine flexibilité quant aux variantes sémantiques supportées :** les profils ne sont pas pris en compte par ce standard, cela limite considérablement la personnalisation du moteur d'exécution pour prendre en charge de nouvelles variantes sémantiques. Pour répondre à ce problème, nous ajoutons les mécanismes nécessaires qui permettent l'application des profils et la capture des extensions sémantiques impliquées par l'utilisation d'un profil. Ces mécanismes exploitent les informations spécifiées dans les stéréotypes afin de paramétrer le moteur d'exécution pour simuler les modèles en fonction d'une sémantique particulière.

Ce manuscrit est organisé en trois parties :

- La première partie délimite le contexte de l'étude et présente les concepts de l'ingénierie dirigée par les modèles (IDM) et les systèmes temps réel embarqués (STRE). Elle adresse ensuite les aspects et les contraintes à prendre en considération pour développer ces systèmes.

- La deuxième partie, caractérise dans un premier temps les différentes sémantiques d'exécutions des langages de modélisation dans le contexte de l'IDM. Cette caractérisation est focalisée sur les différents types de description et leur type d'application. Dans un second temps, nous présentons une analyse de différentes approches et outils permettant la définition des sémantiques et l'exécution des modèles. Cette analyse est réalisée suivant les critères 1, 2, 3 et 4 identifiés ci-dessus afin d'évaluer notre contribution. Nous présentons par la suite en détail le standard fUML de l'OMG en mettant l'accent sur ses limites par rapport aux objectifs de ce travail de thèse.

- La troisième partie décrit les différents volets de la contribution de ce travail.

CHAPITRE 2

Positionnement

1 Ingénierie dirigée par les modèles

Dans cette section nous rappelons les principes fondamentaux de l'ingénierie dirigée par les modèles. Nous mettrons l'accent sur l'approche MDA proposée par l'OMG (Object Management Group), qui constitue un espace technologique et un ensemble de standards pour l'application de l'IDM. L'intérêt de ce rappel est d'introduire le contexte de cette étude. Il vise à caractériser les pratiques de l'approche MDA dans un flot de conception des systèmes à prépondérance logicielle ou matérielle.

1.1 Les principes de l'IDM

L'ingénierie dirigée par les modèles [3] est un paradigme de développement logiciel qui préconise une utilisation intensive et systématique des modèles. Les modèles sont placés au cœur du processus de développement afin de faire face à la complexité croissante de la conception et de la production du logiciel.

L'IDM est distinguée par sa capitalisation du savoir faire non plus au niveau du code source, mais plutôt au niveau des modèles. Cela ne signifie pas une rupture technique avec les solutions de développement existantes telles que les langages de programmation, mais au contraire, elle est complémentaire. En effet, l'utilisation des techniques de transformation de modèles, permet de générer le code source des applications automatiquement ou semi-automatiquement depuis les modèles réalisés lors de la conception.

De nombreux travaux [4, 5, 6, 7] se sont intéressés à cette nouvelle discipline informatique émergente. Ils ont aidé à clarifier et dégager ses concepts. Par la suite, à partir de ces travaux, nous allons définir les notions fondamentales liées à l'IDM.

Nous allons d'abord donner la définition d'un modèle. Nous retiendrons la définition tirée de [8] :

Définition 1 *Un modèle est une abstraction d'un système, construite dans une intention particulière. Cette abstraction devrait être représentative afin de répondre à des questions sur le système modélisé.*

Pour que le modèle soit efficacement manipulable par des outils informatiques, il doit respecter certaines qualités. Dans [9], Bran Selic avait identifié les caractéristiques essentielles qu'un modèle devrait avoir, ce sont principalement :

- ≻ **L'abstraction :** le modèle doit assurer un certain niveau d'abstraction et cacher les détails que l'on ne souhaite pas considérer pendant l'étude.
- ≻ **La compréhensibilité :** le modèle doit être compris intuitivement par les personnes qui l'utilisent. Donc, il doit être exprimé dans un formalisme assez compréhensible dont la sémantique est intuitive.

➢ **La précision :** le modèle doit fournir une représentation fidèle et précise des propriétés du système à étudier.

➢ **L'économie :** le modèle doit rester peu coûteux en comparaison aux coûts du développement du système réel.

Cette liste n'est pas exhaustive et elle peut contenir également d'autres caractéristiques telles que la prédiction, la maintenabilité, la traçabilité, l'exécution, etc. Ainsi, l'obtention de modèles qui respectent toutes ces caractéristiques est une tâche non évidente.

Dans le cadre du développement logiciel, on s'intéresse de plus en plus à la notion de modèles exécutables et interprétables par les machines. Il nous faut donc pouvoir formaliser la syntaxe d'un modèle, c'est-à-dire définir un langage de modélisation. Pour ce faire, l'IDM propose la notion du méta-modèle.

Définition 2 *Un méta-modèle est un modèle d'un langage de modélisation. Il définit les concepts ainsi que les relations entre ces concepts nécessaires à la description des modèles.*

La relation entre un modèle et un méta-modèle est dite relation de conformité. Ainsi, un modèle est conforme à un méta-modèle. Par analogie, nous pouvons comparer cette relation à un concept de la programmation orientée objet, il s'agit d'instance d'objet. Par conséquent, nous pouvons dire qu'un modèle est une instance d'un méta-modèle. Dans l'IDM, le modèle d'un langage de modélisation s'appelle un méta-modèle.

A partir de ces définitions nous identifions deux relations importantes dans l'IDM. La première relation lie le modèle et le système modélisé, elle s'appelle "Représentation de". La deuxième relation lie le modèle et le méta-modèle du langage de modélisation, c'est la relation "Conforme à". La figure 2.1 schématise ces deux relations.

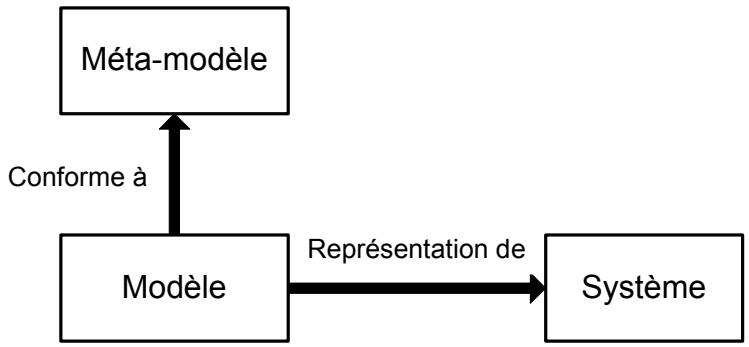


FIGURE 2.1 – Relation de base en IDM

Un autre aspect important dans l'IDM est la transformation de modèles. Cette technique permet le raffinement des modèles, c'est-à-dire le passage d'un modèle abstrait à un modèle plus détaillé en allant jusqu'à la production des artefacts exécutables.

Définition 3 *Une transformation de modèle est une technique qui génère un ou plusieurs modèles cibles à partir d'un ou plusieurs modèles sources conformément à un ensemble de règles de*

transformations. Ces règles décrivent essentiellement comment un modèle décrit dans un langage source peut être transformé en un modèle décrit dans un langage cible.

Les transformations sont dites endogènes quand les modèles manipulés sont issus du même méta-modèle. Quand les modèles sources et cibles sont de différents méta-modèles, les transformations sont dites exogènes. Selon les pratiques de manipulation de modèles dans l'IDM, nous distinguons deux approches de transformation :

- ≻ les transformations de **modèle à modèle** : selon Tom MENS [10], cette catégorie est vue comme un raffinement et amélioration de la qualité du modèle. C'est un passage d'un modèle à un autre modèle du même système dont le méta-modèle peut être différent ou le même. En effet, les transformations de modèles nécessitent l'utilisation de langage de transformations, Jean BÉZIVIN [11] les a classés chronologiquement en trois générations. La première génération est *les langages de transformation de structures séquentielles d'enregistrement*, dans ce cas, un script indique comment le modèle source est transformé en un modèle cible comme les langages de scripts UNIX, AWK ou Perl. La deuxième génération est *les langages de transformation d'arbres*. Les modèles sont représentés dans ce type sous forme d'arbre. La transformation est réalisée par le parcours d'un arbre d'entrée (le modèle source) et la génération d'un arbre de sortie (le modèle cible). Elle se base souvent sur des documents au format XML et l'utilisation de XSLT [12] ou XQuery [13]. La troisième génération forme *les langages de transformation de graphes*, dans cette catégorie le modèle à transformer est représenté par un graphe orienté étiqueté. La transformation est donnée cette fois sous forme d'un modèle conforme à un méta-modèle. ATL [14] est un exemple représentatif de langage de transformation de graphes.

- ≻ les transformations de **modèle à code source** : ou génération de code, il s'agit d'un cas particulier de transformation de modèle à modèle où la grammaire du langage de programmation est le méta-modèle du modèle cible. C'est une représentation textuelle des informations du modèle dans un langage de programmation cible. De plus, elle augmente considérablement la productivité et la portabilité des modèles. Nous trouvons différents langages de génération de code, par exemple XPand [15], Acceleo [16].

La figure 2.2 schématise la technique de transformation de modèles dans le domaine de l'ingénierie dirigée par les modèles.

En résumé, nous avons introduit brièvement les différents concepts fondamentaux de l'ingénierie dirigée par les modèles. Dans la section suivante, nous allons présenter l'approche MDA (Model Driven Architecture), une initiative lancée par l'OMG (Object managment Group) pour la mise en œuvre de l'ingénierie dirigée par les modèles.

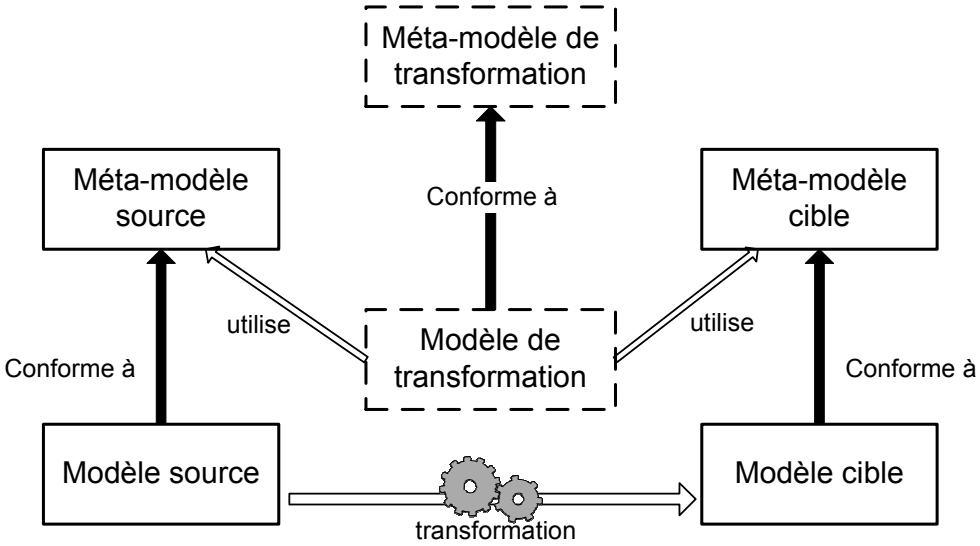


FIGURE 2.2 – La transformation de modèles en IDM

1.2 L'approche MDA

L'approche MDA (Model Driven Architecture) est proposée et soutenue par l'OMG [17] pour le développement des systèmes logiciels. Cette approche fixe les technologies à utiliser pour chaque niveau de modélisation dans l'IDM. Elle repose sur un ensemble de standards de l'OMG dont le standard UML [18] pour l'élaboration de modèles et le standard MOF [19] pour la définition de méta-modèles.

Le principe fondamental du MDA est la séparation entre la logique métier et la logique d'implémentation. Cela est réalisé par la création de modèles du système séparément des plates-formes d'exécution. Cette séparation (entre le modèle du système et de la plate-forme d'exécution) automatise le processus de manipulation de modèles et conduit à la génération automatique du code de l'application.

1.2.1 Démarche MDA

La démarche de développement recommandée par le MDA se divise essentiellement en cinq étapes :

1. L'élaboration du modèle d'exigences ***CIM (Computation Independent Model)***, appelé aussi modèle métier ou modèle du domaine applicatif. Ce modèle aide à maitriser la complexité du système et donne une vision sur des exigences dans un environnement particulier, mais sans rentrer dans le détail de la structure du système, ni de son implémentation.
2. L'élaboration du modèle indépendant de toute plate-forme d'exécution ***PIM (Platform Independent Model)***. C'est un modèle qui ne contient que des informations fonctionnelles du système, sans détails techniques relevant de l'implémentation. À ce

niveau, le formalisme utilisé pour exprimer un PIM est le langage de modélisation UML.

3. L'enrichissement du modèle PIM par raffinement successif. Dans cette étape de nature incrémentale, nous pouvons continuer à détailler le comportement du système tout en omettant les détails liés à la plateforme.
4. Choix des plates-formes d'exécution et génération des modèles correspondants ***PSM (Platform Specific Model)***. Ces modèles sont obtenus par une transformation du PIM en y ajoutant les informations techniques relatives à la plate-forme cible. Ils représentent une vue détaillée du système et sont dépendants d'une plate-forme d'exécution.
5. Les informations liées à la plate-forme d'exécution sont décrites dans un modèle de plate-forme ***PDM (Platfom Description Model)***. Ce modèle fournit les concepts techniques et les services fournis par la plate-forme cible. Notons qu'il existe plusieurs niveaux de détail du PSM. Le premier niveau est issu de la transformation du PIM et du PDM. Le niveau le plus détaillé, est un code généré dans un langage spécifique tel que Java, C++, etc.

En effet, le modèle de chaque étape contient des hypothèses sur la sémantique d'exécution du modèle du système final mais à des niveaux différents d'abstraction. Par exemple, la concurrence peut être décrite à un niveau élevé d'abstraction par des concepts de processus ou d'objet actif [20] [21] puis raffinée en un ensemble de tâches sur une plateforme d'exécution. A un haut niveau l'application est indépendante de la plateforme d'exécution mais à un niveau plus raffiné, elle est spécifique à un système multitâche ou distribué. Ces hypothèses joueront un rôle important dans la simulation du comportement du système final.

1.2.2 Transformation de modèles dans l'approche MDA

Le développement des applications logiciel dans l'approche MDA est centré sur la conception d'un ensemble de modèles et sur leurs transformations. Le formalisme de modélisation UML est ainsi choisi pour exprimer les différents modèles pendant que leurs transformations sont réalisées en utilisant le langage QVT(Query, View, Transformation) [22] standardisé par l'OMG. La figure 2.3 résume les transformations potentielles entre les différents types de modèles que nous avons présentés dans la section précédente.

Les transformations du type CIM vers CIM, PIM vers PIM et PSM vers PSM visent à compléter, spécialiser ou filtrer le modèle. La transformation CIM vers PIM permet de passer de la vue exigence à une vue fonctionnelle préliminaire indépendante de tout détail d'implémentation. La transformation du PIM vers PSM permet de spécialiser le PIM selon la plate-forme d'exécution ciblée en se basant sur les informations fournies par le PDM. La transformation du PSM vers du code est la phase de génération du code, elle permet après

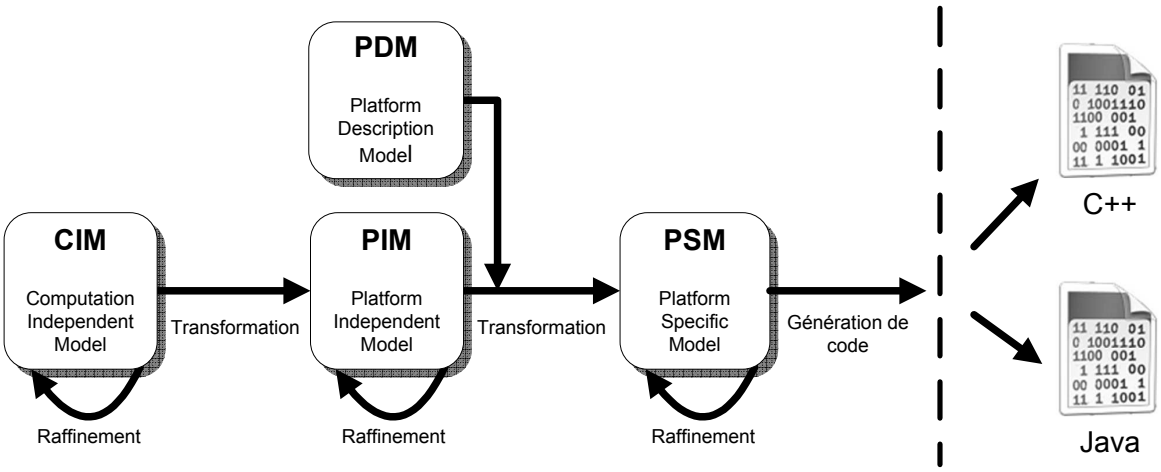


FIGURE 2.3 – Modèles et transformations dans l'approche MDA

compilation d'obtenir l'exécutable de l'application.

1.3 La Sémantique des langages de modélisation

Un langage de modélisation est nécessaire pour l'élaboration et la production des modèles. Dans la sphère de l'ingénierie dirigée par les modèles, ce langage pourrait être un langage à vocation générale ou spécifique. Un langage générique peut être utilisé pour décrire une grande variété de systèmes. Tandis qu'un langage spécifique (DSML pour Domain Specific Modeling langage) contient des concepts distinctifs permettant de décrire des systèmes ciblés. La figure 2.4 récapitule les étapes de création d'un langage de modélisation. Dans les deux cas, la création d'un langage de modélisation consiste à :

1. Définir une syntaxe abstraite du langage.
2. Définir une syntaxe concrète du langage.
3. Définir une sémantique du langage.

1.3.1 La syntaxe abstraite

La syntaxe abstraite d'un langage de modélisation exprime structurellement l'ensemble de ses concepts ainsi que les relations entre ces concepts (présentée par le tableau 2.1). Elle est définie en utilisant un langage de méta-modélisation tel que MetaGME [23] et MOF de l'OMG [19]. Ces langages offrent les concepts et les relations élémentaires permettant la description de la syntaxe abstraite sous forme de méta-modèle. Cependant, les facilités graphiques qu'offrent ces langages manquent d'expressivité pour capturer toutes les propriétés du langage de modélisation, contrairement aux langages de programmation traditionnels où les grammaires hors-contexte [24] sont suffisantes pour exprimer les constructions permises par un langage. Par exemple, supposons que nous voulions définir

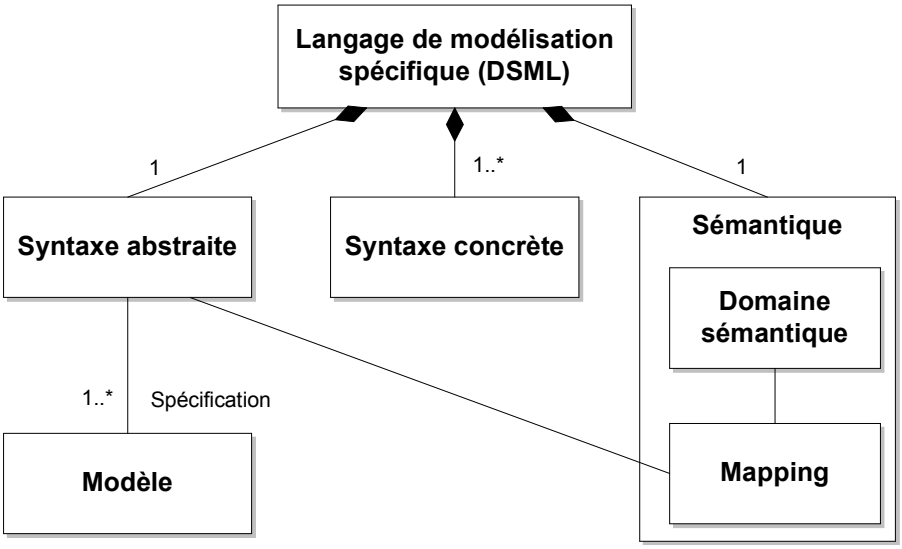


FIGURE 2.4 – Étapes de création d'un langage de modélisation

un langage de modélisation pour les graphes orientés acycliques. En plus des notions de nœuds et d'arêtes, une contrainte supplémentaire s'impose qui interdit les cycles dans un graphe. Cette contrainte n'est pas exprimable en langage graphique comme MOF sans l'utilisation d'un langage de contraintes supplémentaires tel que OCL [25]. Dans le contexte des langages de modélisation, ces contraintes sont spécifiées au niveau du méta-modèle sous forme de règles de bonne formation qui devront être respectées par les modèles conformes à ce méta-modèle.

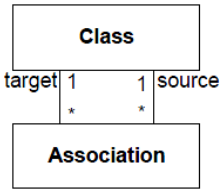
Type de langage	**Syntaxe abstraite définie par**	**Exemple**
Langage naturel	Une description	La proposition relative est une subordonnée introduite par un pronom relatif
Langage de programmation	Une description MétaNot (systèmes à base de productions)	AST (Arbre de syntaxe abstraite)
Langage de modélisation	Un méta-modèle	

TABLE 2.1 – Aperçu des différentes syntaxes abstraites

1.3.2 La syntaxe concrète

La syntaxe concrète fournit aux utilisateurs les notations nécessaires (présentées par le tableau 2.2) pour exprimer un modèle. Les langages de programmation traditionnels ont généralement une notation textuelle, tandis que les langages de modélisation peuvent être textuels ou visuels. La syntaxe concrète permet de manipuler les concepts de la syntaxe abstraite et créer des modèles qui lui sont conformes. Elle est définie en annotant chaque concept de la syntaxe abstraite par une décoration qui sera manipulée par un utilisateur potentiel du langage. Il existe plusieurs outils qui permettent de définir des syntaxes concrètes, nous citons principalement GMF [26] pour des syntaxes graphiques et TCS [27] pour des syntaxes textuelles.

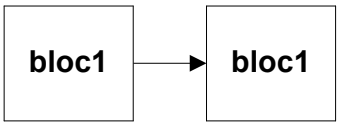
Type de langage	**Syntaxe concrète définie par**	**Exemple**
Langage naturel	"Voiture", "Maison"	Une liste de mots
Langage de programmation	Une liste de mots-clés, des expressions régulières	While, if .. else, etc.
Langage de modélisation	Un ensemble de blocs et de connecteurs	

TABLE 2.2 – Aperçu des différentes syntaxes concrètes

1.3.3 La sémantique

La sémantique d'un langage de modélisation décrit précisément et sans ambigüité la signification des concepts de ce langage. On dit qu'une sémantique est formelle lorsqu'elle est exprimée dans un formalisme mathématique. Elle permet de donner d'une manière claire et rigoureuse le sens d'un modèle. Une sémantique est alors caractérisée par [28] :

- ≻ Un domaine sémantique dont les concepts sont compréhensibles et bien définis.
- ≻ Un mapping entre les éléments de la syntaxe abstraite et les éléments du domaine sémantique.

Prenons par exemple le langage de programmation C. La sémantique d'un programme est donnée par l'exécution d'un ensemble de mots-clés qui revient à exécuter un jeu d'instructions d'une machine cible. Le domaine sémantique dans ce cas est constitué d'un jeu d'instructions de la machine cible et le mapping sémantique est réalisé par un compilateur qui traduit des programmes écrits en C dans le jeu d'instructions approprié.

La sémantique d'un langage de modélisation peut être divisée en deux types : la sémantique statique ou structurelle et la sémantique dynamique ou comportementale. La sémantique statique traite des propriétés à vérifier avant l'exécution et concerne la

signification du modèle en termes de structure et de règles de bonne formation. Quant à la sémantique dynamique, elle permet de donner une description du comportement du modèle pendant l'exécution. Traditionnellement, la sémantique de nombreux langages de modélisation est décrite de manière informelle, en langage naturel, nous citons notamment la norme UML2 [18]. Une sémantique informelle pose en fait plusieurs problèmes, nous identifions :

- L'ambigüité et les mauvaises interprétations des concepts du langage de modélisation par les utilisateurs. Les normes des langages exigent une sémantique précise. Une norme d'un langage mal définie sémantiquement est ouverte à l'incompréhension et aux mauvaises utilisations par les développeurs d'outils tout en limitant significativement l'interopérabilité des modèles.
- Une sémantique informelle ne peut être interprétée par les outils, ce qui incite les développeurs d'outils à implémenter leur propre compréhension de la sémantique. Le même langage pourrait être implémenté différemment. Ainsi, deux outils différents peuvent offrir des implémentations contradictoires de la même sémantique.

En effet, associer une sémantique précise au langage apporte une solution aux problèmes précédemment cités. De plus, elle est essentielle pour centraliser le sens des modèles entre les différents acteurs du processus de développement. Cela est particulièrement important dans le cas des modèles destinés aux systèmes temps réel et embarqués. Ainsi, la sémantique joue un rôle capital pour déterminer les capacités essentielles d'un langage dans une démarche de développement IDM tel que la capacité d'exécution, de transformation ou de vérification.

En résumé, ce chapitre a présenté le contexte général de ces travaux de thèse qui s'inscrivent dans le cadre du développement des systèmes temps réel embarqués. Dans la suite nous nous intéressons à la modélisation de la partie applicative de ces systèmes dans une architecture monoprocesseur. Nous focalisons principalement notre étude sur les aspects de concurrence entre tâches et le respect des contraintes temporelles. Dans le chapitre suivant nous allons présenter un état de l'art portant sur la modélisation de la partie applicative en mettant en évidence les sémantiques d'exécution des langages de modélisation.

1.4 Langage de modélisation : UML

L'approche MDA préconise l'utilisation du langage de modélisation UML (Unified Modeling Language) [18] pour la construction de modèles tout au long du cycle de développement du logiciel. C'est un langage de modélisation graphique à caractère générique, utilisé pour spécifier, visualiser et documenter les artefacts d'un système logiciel.

Le langage UML est défini sous forme d'un méta-modèle, organisé en plusieurs paquetages. Chaque paquetage introduit des concepts que nous pouvons exprimer par des notations graphiques à travers des diagrammes. Ces diagrammes peuvent être regroupés

en deux classes principales : celle relative à la modélisation structurelle et celle relative à la description du comportement. La figure 2.5 décrit l'organisation de ces différents diagrammes. Nous donnons ci-dessous une courte description de certains diagrammes.

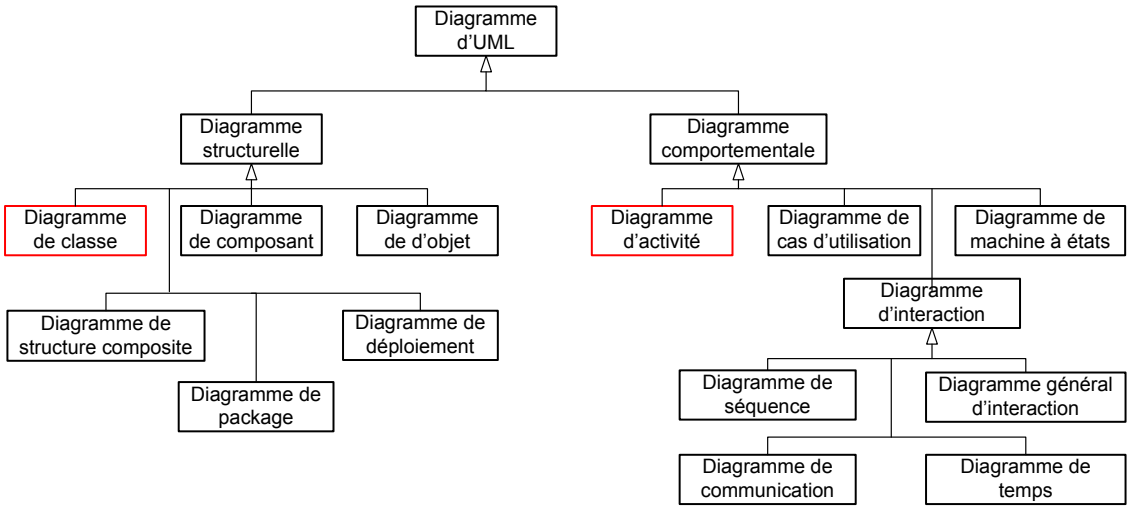


FIGURE 2.5 – Organisation des diagrammes d'UML

➢ **Le diagramme de classe :** ce diagramme couvre un aspect fondamental dans la modélisation, il permet d'exprimer d'une manière précise la structure statique d'un système, en termes de classes et de relations entre ces classes. Une classe correspond à la description d'un concept du domaine d'application considéré.

➢ **Le diagramme d'objet :** est destiné à représenter les instances des classes définies dans le diagramme de classe et les liens qui les connectent. Ce diagramme fige ainsi une image du système modélisé à un instant donné.

➢ **Le diagramme de composant :** manipule la notion de composant qui permet de représenter une abstraction d'un ensemble de classes réalisant différentes fonctionnalités et qui communiquent par des interfaces avec leur environnement. Le diagramme de composant permet d'expliciter les unités logicielles qui forment le socle du système.

➢ **Le diagramme de structure composite :** ce diagramme manipule la notion de port et de connecteur. Il permet d'exprimer la composition entre les classes et leurs éléments internes.

➢ **Le diagramme de cas d'utilisation :** capture les fonctionnalités que nous offre un système. Il manipule les concepts d'acteur et de cas d'utilisation. Un acteur est une entité extérieure du système qui peut initier l'un des cas d'utilisation. Ce diagramme décrit, sous forme d'actions et de réaction le comportement d'un système du point de vue utilisateur. Souvent, il est utilisé dans les premières phases de spécification.

➢ **Le diagramme d'activité :** permet de se focaliser sur la partie dynamique d'un système. Il permet de définir le comportement d'une opération ou le déroulement d'un

cas d'utilisation, par l'enchaînement et la coordination d'un ensemble d'actions. Une action est une étape de calcul élémentaire. L'enchainement d'un ensemble d'actions peut être représenté par des flots de contrôle ou des flots de données ou bien des deux.

- **Le diagramme de machine à états :** permet de décrire les comportements discrets d'un système via des automates d'états finis. Il s'agit de graphes d'états, reliés par des arcs orientés qui décrivent les transitions. Ce diagramme permet de décrire les changements d'états d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets/composants ou avec des acteurs.

- **Le diagramme de séquence :** illustre la séquence d'interactions entre objets selon l'ordre chronologique des envois de messages entre eux.

1.5 Extension du langage UML

L'objectif d'UML en proposant autant de diagrammes et concepts est de fournir un langage de modélisation généraliste qui couvre une grande partie du domaine du logiciel. Cependant, il arrive parfois que ce langage ne soit pas adapté à capturer certaines propriétés particulières à un domaine spécifique. C'est le cas par exemple des systèmes temps réel, pour lesquelles il est nécessaire de spécifier des propriétés temporelles (échéance, période, etc.), ce qui n'est pas supporté par la notation standard d'UML. Pour remédier à ce problème, il existe deux solutions pour étendre UML.

La première méthode d'extension est nommée extension lourde (*Heavyweight extension*). Elle est fondée sur la manipulation d'une copie du méta-modèle d'UML en lecture et en écriture. En d'autres termes, ce mécanisme permet d'importer des éléments du méta-modèle d'UML en utilisant l'opération *merge* [29] et d'ajouter, supprimer ou modifier certaines caractéristiques. Toutes les modifications sont alors effectuées directement sur une copie des méta-classes concernées.

La deuxième méthode, est dite extension légère (*Lightweight extension*), dans ce cas, le méta-modèle d'UML n'est manipulé qu'en lecture et les éléments importés sont utilisés sans modification de leurs caractéristiques. Cette technique est réalisée grâce au concept de profil, normalisé par l'OMG [18]. Un profil permet alors d'étendre ou de restreindre le méta-modèle d'UML afin de l'adapter à un domaine spécifique. Il offre les moyens pour réaliser les modifications suivantes sur le méta-modèle d'UML :

- étendre les méta-classes du méta-modèle UML via des stéréotypes.

- ajouter des propriétés aux stéréotypes.

- spécifier de nouvelles règles de bonne formation concernant les stéréotypes sous forme de contraintes OCL.

La figure 2.6 montre un fragment du méta-modèle UML correspondant aux éléments de profil. Ce dernier est constitué fondamentalement de stéréotypes, de valeurs étiquetées et de contraintes.

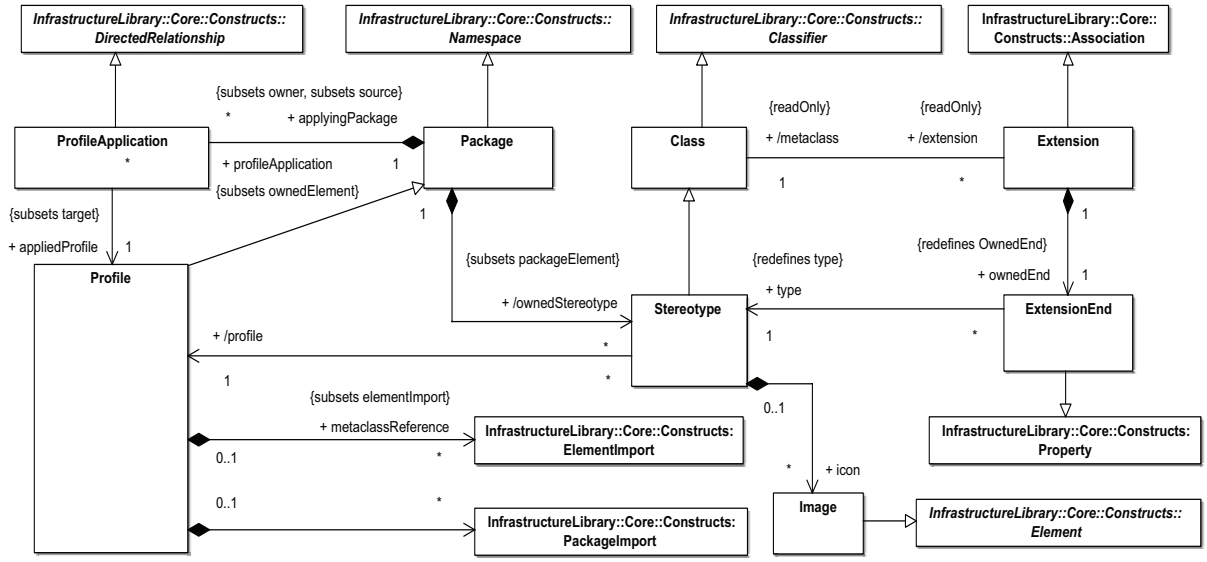


FIGURE 2.6 – Les éléments du paquetage profile (source OMG)

Le stéréotype est le mécanisme de base pour étendre UML. Il étend une ou plusieurs méta-classes existantes du méta-modèle UML afin de permettre leur utilisation dans un domaine spécifique. Quant aux valeurs étiquetées, elles sont caractérisées par un nom et un type et sont considérées comme les propriétés des stéréotypes. En effet, un stéréotype est défini au niveau M2 et sera appliqué à un élément du modèle (niveau M1) et les valeurs de ses propriétés représentent les valeurs étiquetées. En plus des stéréotypes et des valeurs étiquetées, des contraintes peuvent être spécifiées afin de contraindre la sémantique des éléments du méta-modèle. Les contraintes peuvent être non formelles et renseignées dans la documentation du profil ou bien formelles et décrites en langage de contraintes OCL(Object Constraint Language) [25],standardisé par l'OMG.

L'OMG a standardisé plusieurs profils couvrant différents domaines, nous en mentionnons quelques-uns :

1. Le profil CORBA CCM [30], pour la modélisation d'applications à base de composants.
2. Le profil SoC (System on a Chip) [31], offre des facilités pour la modélisation des systèmes sur puce.
3. Le profil SPT (Schedulability, Performance and Time) [32], pour la modélisation d'applications temps réel .
4. Le profil UTP (UML Testing) [33] Profile qui permet la spécification des tests d'application.

L'OMG a standardisé plusieurs profils pour la modélisation des applications temps-réel embarquées. On s'intéressera particulièrement au profil MARTE (Modeling and Analysis of Real-time and Embedded Systems)[34] que nous allons utiliser le long cette étude. Ce profil définit les fondements pour la description et l'analyse des systèmes temps réel et

embarqués (STRE). MARTE se concentre principalement sur l'analyse d'ordonnancabilité et de performance. Il offre aussi les concepts qui permettent de modéliser les caractéristiques matérielles des STRE. La partie modélisation du profil fournit le support nécessaire partant de la spécification à la conception détaillée des caractéristiques temps réel et embarquées des systèmes. La partie analyse fournit des mécanismes permettant d'annoter les modèles avec les informations requises pour effectuer des analyses spécifiques. Le profil MARTE remplace le profil SPT.

2 Développement des systèmes temps réel embarqués (STRE)

2.1 Introduction aux systèmes temps réel embarqués

Les systèmes informatiques temps réel et embarqués sont présents de plus en plus dans plusieurs domaines d'application. Parmi ceux-ci nous citons par exemple : le domaine du transport (automobile, ferroviaire, aéronautique), le domaine militaire (missiles, radars), le domaine des télécommunications (téléphonie, serveurs, box), le domaine de l'électroménager (téléviseurs, machines à laver), le domaine du multimédia (consoles de jeux, navigateurs GPS), etc.

Un système est qualifié comme temps réel [35] si son bon fonctionnement est caractérisé non seulement par la correction des valeurs qu'il produit, mais aussi par le respect de ***contraintes temporelles*** sur cette production de valeurs. Un système temps réel interagit avec son environnement, généralement caractérisé par un procédé physique, qui lui-même évolue avec le temps. Le temps de réponse désiré d'un tel système est alors fixé par l'environnement avec lequel il interagit. De plus, ce type de système réagit en permanence aux variations du procédé (l'environnement) et agit en conséquence pour obtenir le comportement souhaité, cette caractéristique définit la notion de réactivité. La définition suivante explique le fonctionnement d'un système temps réel :

Définition 4 *Un système réactif est un système qui réagit continuellement avec son environnement à un rythme imposé par cet environnement. Il reçoit, par l'intermédiaire de capteurs, des entrées provenant de l'environnement, appelées stimuli. Il réagit à tous ces stimuli en effectuant un certain nombre d'opérations et il produit, grâce à des actionneurs, des sorties utilisables par l'environnement, appelées réactions ou commandes.*

La figure 2.7 illustre la définition précédente. Un système temps réel est alors constitué de deux sous systèmes distincts : le procédé à contrôler et le système de contrôle. Le procédé contient des capteurs et des actionneurs. Les capteurs vont récupérer des informations du procédé et les transmettre au système de contrôle. Quant aux actionneurs, ils vont commander le procédé afin de réaliser une tâche bien définie. Le système de contrôle est composé d'un calculateur et d'interfaces d'entrées/sorties qui servent à communiquer avec le procédé. Le calculateur va exécuter un algorithme de contrôle en respectant des

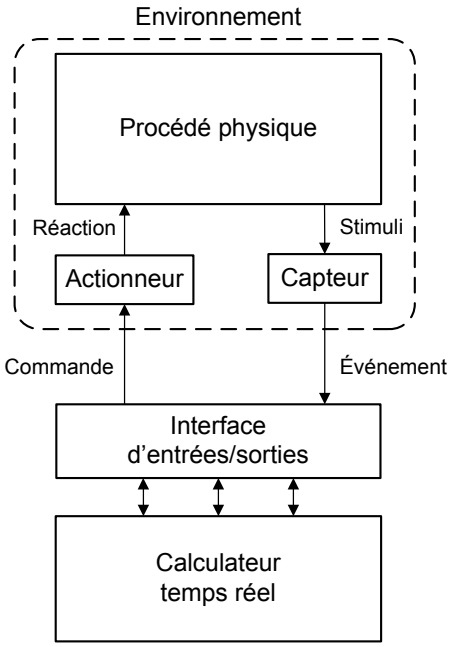


FIGURE 2.7 – Vue d'ensemble d'un système temps réel

propriétés temporelles et envoyer des ordres de commande aux actionneurs via l'interface de communication. Il est important de noter qu'un système temps réel est dit ***embarqué*** lorsqu'il est enfoui à l'intérieur de l'environnement avec lequel il interagit, comme un calculateur dans un avion ou une voiture.

2.2 Classification

La classification la plus répandue des systèmes temps réel est celle basée sur le type de contraintes temporelles à respecter [36], nous distinguons alors :

- **Systèmes temps réel à contraintes strictes :** la réponse du système dans les délais est vitale, toutes les contraintes temporelles doivent être impérativement respectées. L'absence de réponse à temps n'est pas tolérée et peut provoquer des conséquences catastrophiques [37].À titre d'exemple nous citons les systèmes de contrôle de vol, les systèmes de contrôle de station nucléaire, etc.

- **Systèmes temps réel à contraintes souples :** à la différence des systèmes durs, la réponse du système après les délais réduit progressivement son intérêt, le non-respect des contraintes temporelles est toléré et les pénalités ne sont pas catastrophiques [38]. Nous citons à titre d'exemple les applications multimédias.

En réalité, la plupart des systèmes temps réel sont hybrides. C'est-à-dire ils contiennent les deux types de contraintes temporelles strictes et souples. D'autre part, les systèmes temps réel sont considérablement influencés par le mode d'exécution adopté [39], il existe deux modes : ***l'exécution dirigée par les évènements (event-triggered)*** et ***l'exécution***

dirigée par le temps (time-triggered). Un système dirigé par les évènements suit un principe de réaction à la demande. Le système réagit suivant les évènements qu'il reçoit. L'interaction avec l'environnement dans ce mode est relativement imprévisible impliquant des problématiques particulières de dépassement de contraintes temporelles et empêche de fournir des spécifications temporelles précises sur le système. Les systèmes dirigés par le temps sont basés sur la progression globale du temps pour réaliser le contrôle du système. Généralement les composants du système dans ce cas sont périodiques et notamment ceux en interaction avec l'environnement. Adopter ce mode d'exécution permet de spécifier précisément le comportement temporel du système.

2.3 Architecture

Un système temps réel est une association d'une ***couche logicielle*** et une ***couche matérielle*** où le logiciel permet une gestion adéquate des ressources matérielles en vue de remplir certaines fonctions dans des limites temporelles bien précises. La figure 2.8 illustre une architecture d'un système temps réel embarqué. Dans le cas d'un système distribué une couche logicielle de gestion telle qu'un intergiciel (middleware), pourrait être ajoutée.

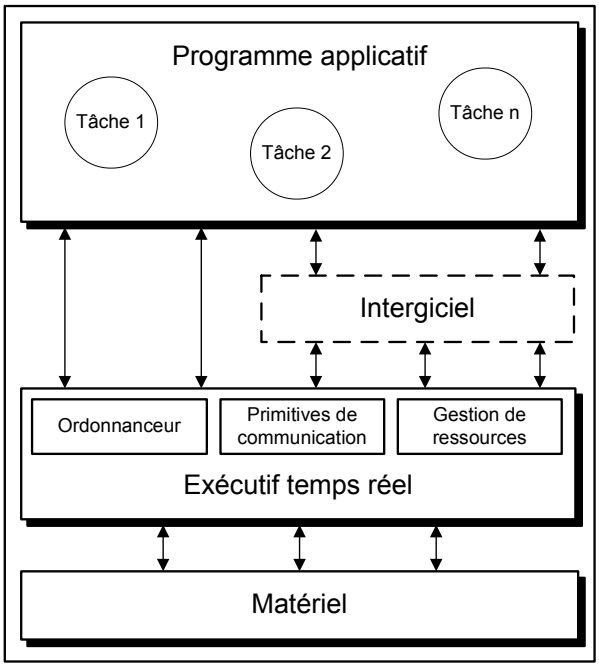


FIGURE 2.8 – Architecture d'un système temps réel embarqué

2.3.1 La couche matérielle

Cette couche regroupe l'ensemble des ressources physiques nécessaires à l'exécution de la couche logicielle qui pilote le procédé. Cela inclut les processeurs, les mémoires, les cartes d'entrés/sorties (capteurs/actionneurs), les réseaux, etc. La couche matérielle peut

être classée en fonction du nombre de processeurs utilisés et la présence éventuelle d'un réseau en trois catégories d'architecture :

➢ **Architecture monoprocesseur :** dans ce cas, la couche matérielle est constituée d'un seul processeur, le temps processeur est donc partagé entre les différentes tâches.

➢ **Architecture multiprocesseur :** la couche matérielle est composée de plusieurs processeurs qui partagent une mémoire centrale. Les applications du système dans ce cas sont réparties sur les différents processeurs.

➢ **Architecture distribuée :** la couche matérielle est composée de plusieurs processeurs dont chacun possède sa propre mémoire. Les processeurs dans ce cas sont reliés les uns aux autres par un réseau. Les applications du système sont réparties sur les différents processeurs et communiquent entre elles via le réseau.

2.3.2 La couche logicielle

Elle est composée principalement de deux parties. La première partie est de bas niveau. Elle est représentée par l'exécutif temps réel (OS temps réel) qui fait le lien avec la couche matérielle. La deuxième partie est de haut niveau, elle correspond au programme applicatif de contrôle.

- **L'exécutif temps réel :** ou système d'exploitation temps réel est composé d'un noyau temps réel et de modules ou bibliothèques complétant le noyau et facilitant le développement de l'application. Un noyau doit répondre à certaines exigences pour être caractérisé de temps réel et fournir notamment des services de base en l'occurrence : un service d'ordonnancement, un service de gestion de ressources et un service fournissant des primitives de communication. Le rôle principal de l'exécutif temps réel est d'ordonnancer les exécutions des tâches de l'application de contrôle. En plus de ces services, l'exécutif doit fournir des garanties concernant le temps d'exécution nécessaire à ses primitives afin de permettre d'estimer le temps d'exécution nécessaire à chaque tâche du système temps réel. Nous citons quelques exemples d'exécutif : VxWorks [40], RT Linux [41], Osek/VDX [42].
- **L'applicatif de contrôle :** ou le programme informatique, correspond à la partie logicielle du système qui va réaliser les différentes fonctions de contrôle du procédé. Ce programme est composé d'entités de base appelées tâches. Chaque tâche assure une fonction qui lui est propre en utilisant des routines mises à disposition par l'exécutif temps réel comme l'acquisition de données en provenance des capteurs. Une tâche temps réel est caractérisée par la définition de contraintes temporelles. La figure 2.9 représente les caractéristiques temporelles les plus communes :

  ➢ r_i (Ready time) représente la date d'activation de la tâche.

  ➢ S_i (Start time) est la date de démarrage d'une tâche. C'est la date à laquelle elle commence effectivement son exécution

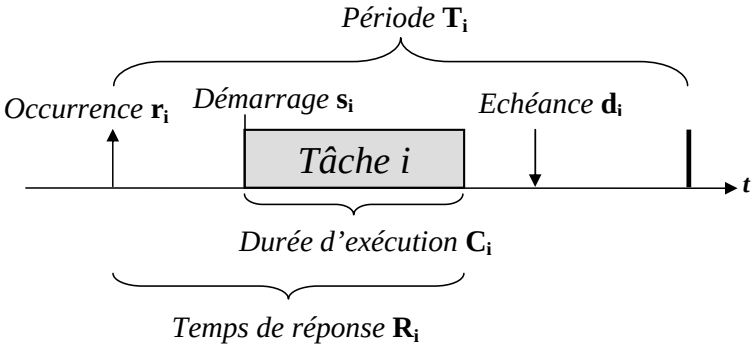


FIGURE 2.9 – Modèle de tâche

➢ C_i représente la durée d'exécution de la tâche.

➢ T_i correspond à la période d'activation de la tâche, c'est-à-dire l'intervalle de temps séparant deux occurrences successives de la tâche.

➢ D_i (deadline) désigne l'échéance d'une tâche (deadline). C'est la date à laquelle l'exécution de la tâche doit être terminée.

➢ R_i est le temps de réponse de la tâche.

Les tâches temps réel peuvent être classées selon leurs caractéristiques temporelles en différentes catégories [43], nous différencions alors les tâches périodiques, les tâches apériodiques et les tâches sporadiques. Les tâches périodiques représentent des tâches récurrentes dont les activations successives sont séparées par un délai constant. Elles sont caractérisées par un quadruplet (r,C,R,T) issu de [44]. Les tâches apériodiques ont une date d'activation qui n'est pas précise et qui est généralement déclenchée par un évènement extérieur. Le doublet (r,C) suffit pour représenter ce type de tâches (tâches apériodiques). Quant aux tâches sporadiques, elles sont récurrentes mais la durée séparant deux activations successives varie tout en respectant une durée minimale connue à priori.

2.4 Processus de développement

Il existe essentiellement deux processus de développement : le processus de développement ***conjoint ou co-design*** [45] dans lequel les parties logicielles et matérielles sont développées simultanément et le processus de développement ***séparé*** [46] qui consiste à développer les composantes logicielle et matérielle séparément. Les deux processus partagent globalement les différentes phases de développement identifiées dans [47]. Ces différentes phases sont :

➢ **La phase d'analyse :** qui consiste à identifier les principaux caractéristiques de toutes les solutions possibles pour concevoir le système désiré.

➢ **La phase de conception :** c'est dans cette phase que nous ajoutons des éléments

nécessaires à la conception pour définir une solution particulière qui optimise certains critères.

- **La phase d'implantation :** consiste à réaliser ce que nous avons défini dans la phase de conception et fournir, automatiquement ou manuellement, le code source de la partie logicielle du système et les composants de la partie matérielle.
- **La phase de test :** c'est la phase où nous vérifions que l'implantation correspond bien à la spécification et respecte tous les critères identifiés dans la phase d'analyse.

À noter que dans l'approche de développement séparée, l'intégralité des fonctionnalités du système sont des algorithmes dans la partie logicielle, une fois développée suivant ces différentes phases, l'application finale est générée pour une plateforme matérielle, généralement constituée de composants matériels à vocation générale. Néanmoins, dans l'approche de développement conjointe, une phase indispensable s'ajoute aux différentes phases, c'est l'étape de partitionnement logicielle-matérielle, pendant laquelle le choix d'implémentation des fonctionnalités du système est effectué entre la partie logicielle ou matérielle.

2.5 Conclusion

Nous avons introduit dans les sections précédentes, les caractéristiques principales des systèmes temps réel embarqués. En résumé, les STRE interagissent en permanence avec leur environnement afin de réaliser une tâche bien définie. Ils sont composés de deux couches matérielle et logicielle. La couche logicielle doit gérer efficacement la couche matérielle pour répondre aux stimulis de l'environnement naturellement concurrents et contraints par le temps. Le développement des STRE doit prendre en compte l'ensemble des exigences suivantes :

- **Caractéristique d'embarquabilité :** un STRE doit répondre à différentes contraintes au niveau des ressources, en particulier, la taille de mémoire et la consommation d'énergie. Il faudra s'assurer que le système ne consomme pas plus de ressources que celles à sa disposition.
- **Caractéristique de temps :** un STRE doit répondre à des contraintes temporelles, dans ce cas, le respect des échéances doit être vérifié.
- **Caractéristique de concurrence :** un STRE est souvent décomposé en plusieurs tâches, qui sont exécutées en parallèle. Il faudra fournir les mécanismes de communication, de synchronisation et d'ordonnancement nécessaires aux des différentes tâches.

La prise en compte de toutes ces exigences par les processus de développements traditionnels devient de plus en plus difficile à cause de la complexité croissante des STRE (toujours plus de fonctionnalités à intégrer face à des contraintes impondérables de temps de

mise sur le marché). Pour cela, l'industrie des STRE identifiée dans l'IDM, une opportunité pour maitriser cette complexité et rationaliser les flots de conception. La section suivante présente comment l'IDM adresse le développement des STRE.

2.6 Ingénierie dirigée par les modèles pour les systèmes temps réel embarqués

Le développement des systèmes temps réel embarqués nécessite des spécifications précises, complètes et non ambiguës afin d'obtenir des logiciels sûres, fiables et efficaces. L'ingénierie dirigée par les modèles apporte de bonnes pratiques pour le développement de logiciels. Ces pratiques serviront également aux applications temps réel embarquées pour plusieurs raisons [48] :

- ≻ La spécification des applications temps réel embarquées comprend différents points de vue (ex. fonctionnel, temps-réel, tolérance aux fautes, etc.). Cela nécessite des techniques d'abstraction lors du développement.
- ≻ Les options d'implémentation visées peuvent varier considérablement ; différents modèles d'exécution peuvent être envisagés pour un même modèle en fonction de contraintes de réalisation particulières (modèle multi-tâches, communication synchrone, programmation en boucle, etc).
- ≻ La contrainte de performance des STREs s'oppose aux techniques standard du développement logiciel. Les optimisations potentiellement réalisées peuvent produire un code fonctionnel qui pénalise la maintenabilité de l'application finale.
- ≻ Le test et la validation des applications temps réel embarquées sont des activités critiques qui nécessitent la mise en place de modèles et d'outils d'analyse sophistiqués et spécifiques.

L'utilisation de l'IDM pour le développement des STREs, nécessite la définition d'un ensemble cohérent et complet d'artéfacts (règles méthodologiques, transformations de modèles, génération automatique de code) préalablement testés, outillés et évalués. Il reste néanmoins que l'ingénierie dirigée par les modèles présente un inconvénient mineur qui consiste à produire un code généré moins performant qu'un code optimisé directement pour une plateforme cible.

CHAPITRE 3

Etat de l'art

Dans ce chapitre nous positionnons les travaux de cette thèse par rapport aux approches existantes. Il est divisé en trois sections. La première section s'intéresse aux mécanismes de définition des sémantiques des langages de modélisation dans le contexte de l'IDM, elle présente les déférents types de définition de sémantique et donne les avantages et les inconvénients de chaque type de sémantique par rapport au type d'application (exécution, analyse, vérification, etc.). La deuxième partie étudie plusieurs approches et outils permettant la définition des sémantiques et l'exécution des modèles. Nous analysons et évaluons les différentes approches selon des critères que nous avons identifiés dans l'introduction. Dans la troisième section, nous présentons les concepts fondamentaux de la norme OMG « Semantics of a Foundational Subset for Executable UML Models » qui fournit une base nécessaire à nos travaux. Cette norme s'intéresse particulièrement à la sémantique d'exécution des modèles UML. Nous décrivons notamment ses limitations vis-à-vis des besoins d'exécution des modèles temps réel embarquées sur lesquelles nous apportons des réponses.

1 Mécanismes de définition des sémantiques : Classification

Il existe plusieurs approches pour définir la sémantique d'un langage de modélisation. Dans le contexte de l'IDM la sémantique d'un langage est définie au niveau du méta-modèle. Dans ce qui suit, nous allons présenter les approches clefs en illustrant leur utilisation. Nous donnons un aperçu sur les techniques qui permettent de définir des sémantiques pour exécuter un modèle en vue de l'animer ou le simuler. Trois grands types de formalismes sont identifiés dans [49] pour décrire des sémantiques de langages. Selon les besoins d'utilisation du langage tel que la vérification, la transformation ou l'animation de modèles, chaque type de formalisme a ses propres avantages et inconvénients.

1.1 Les sémantiques axiomatiques

L'approche axiomatique [50] est basée sur la logique mathématique. La sémantique dans ce cas est spécifiée par des assertions (ou axiomes) sur des propriétés des éléments syntaxiques du langage. Une assertion est généralement exprimée par des triplets de Hoare [51] {P} S {Q}, dont P est une pré-condition, S est un élément syntaxique du langage, et Q une post-condition. Dans le cadre d'un langage de modélisation, la sémantique axiomatique peut être exprimée sur la syntaxe abstraite soit par le langage de méta-modélisation lui-même, c'est le cas des multiplicités, soit par un autre langage tel que OCL [25]. L'approche axiomatique est la plus abstraite des 3 méthodes. Elle est principalement utile pour prouver certaines propriétés statiques de la sémantique des modèles. Néanmoins, il reste difficile de décrire du comportement avec ce type de sémantique [49].

1.2 Les sémantiques dénotationnelles

La sémantique dénotationnelle a été introduite par Scott et Strachey [52] [53]. Son principe est d'associer chaque concept du langage d'origine à un objet mathématique appelé dénotation d'un autre formalisme rigoureusement défini. On parle aussi de modèle *formel* du langage. Ces objets forment le domaine sémantique du langage d'origine alors que l'association définit sa sémantique.

Par ailleurs, la difficulté que nous pouvons rencontrer pour décrire ce type de sémantique est d'identifier les objets mathématiques qui correspondent pertinemment aux concepts du langage et de créer des correspondances entre eux. Dans le contexte de l'IDM, la sémantique d'un langage de modélisation est définie par sa transformation ou traduction vers un autre langage formellement définit tel que les réseaux de Pétri [54]. Sachant que les réseaux de Pétri ont une sémantique formelle, le modèle obtenu prend donc le sens du réseau de Pétri généré. En pratique, cela revient à créer une passerelle entre l'espace technique source et cible permettant en conséquence de profiter des outils disponibles dans l'espace technique cible pour des fins de simulation ou de vérification par exemple.

1.3 Les sémantiques opérationnelles

La sémantique opérationnelle [55] décrit comment les programmes ou les modèles sont directement exécutés sur une machine réelle ou abstraite, ce qui revient à décrire une forme d'interpréteur pour le langage. Contrairement à la sémantique dénotationnelle qui s'exprime sur une représentation abstraite différente de celle définie pour le langage considéré, la sémantique opérationnelle s'exprime directement sur la même représentation abstraite du langage. Elle est plus facile à comprendre et à écrire. Cela permet d'implémenter facilement des interpréteurs de langages et de mettre et mettre en œuvre les outils support pour l'exécution.

Dans le cadre de l'IDM, la sémantique opérationnelle permet d'exprimer le comportement de la syntaxe abstraite afin d'exécuter les modèles qui lui sont conformes. Elle consiste à enrichir la syntaxe abstraite par des opérations caractérisant le comportement de chaque concept d'une manière impérative et décrivant l'évolution du modèle. Concrètement, cela pourrait être réalisé en utilisant un langage d'action comme AS-MOF [57] ou un langage de méta-programmation tel que Kermeta [58].

Il est à noter que récemment, d'autres classifications ont été proposées. Hausmann [61] a proposé dans sa thèse une classification des techniques pour l'expression de la sémantique comportementale. La sémantique comportementale dans ce cas est spécifiée en fonction d'objectifs particuliers tels que la vérification de propriété et l'analyse de cohérence et la génération de code. Hausmann s'est intéressé particulièrement aux langages de modélisation visuelle sur lesquels il a expérimenté la notion de la méta-modélisation dynamique (DMM), proposée dans sa thèse, comme technique de description

de sémantique.

Par ailleurs, les auteurs de [62] ajoutent aux différents formalismes de description de sémantique mentionnés précédemment, deux autres approches. Ces approches sont :

- **Approche par traduction :** Cette approche décrit la sémantique d'un langage en termes de concepts primitifs d'un autre langage possédant une sémantique opérationnelle bien définie.Elle ressemble à l'approche dénotationnelle sauf que le langage cible de traduction doit avoir une sémantique opérationnelle. Par exemple, traduire le méta-modèle de la syntaxe abstraite d'UML en un autre méta-modèle d'un langage bien définit tel que XCore [63].
- **Approche par extension :** Cette approche consiste à étendre les concepts et la sémantique d'un langage existant. Les concepts du nouveau langage de modélisation héritent leur sémantique des concepts d'un autre langage. L'avantage de cette approche est la réutilisation des sémantiques complexes sans efforts considérables. Cette approche peut être appliquée sur des langages dont la sémantique est une sémantique axiomatique, dénotationnelle ou opérationnelle.

1.4 Comparaison entre les différents types de sémantiques

Le choix d'un mécanisme de définition de sémantique dépend fortement des besoins du langage à définir. Le tableau 3.1 donne une étude comparative entre les différents types de sémantiques.

En résumé, la sémantique est cruciale pour comprendre et définir le sens d'un langage de modélisation. Nous avons présenté les différents styles qui permettent la formalisation des sémantiques. Nous nous focalisons dans la suite sur les sémantiques d'exécution permettant de donner un caractère opératoire aux modèles. Nous proposons dans la section suivante une étude des différents outils et approches qui s'intéressent à la définition des sémantiques opérationnelles pour les systèmes temps réel embarqués.

Type de sémantique	Avantage	Inconvénient
Informelle	– Facile à communiquer. – Ne nécessite pas un apprentissage.	– Ambigüe. – Contient des incohérences. – Ne peut pas être formellement analysée.
Axiomatique	– Fournit une définition mathématique de la sémantique. – Les axiomes sont concis et compréhensibles. – Permet l'utilisation de méthodes mathématiques telles que les preuves et le model-checking.	– La description devient importante et complexe quand plusieurs concepts sont pris en compte.
Dénotationelle	– Donne une description formelle de la sémantique. – S'appuie sur un domaine sémantique bien défini. – Bien adaptée pour le raisonnement mathématique et l'analyse formelle de propriétés.	– Assez complexe pour un utilisateur final. – L'expression d'états et d'opérations est parfois difficile.
Opérationnelle	– Fournit une bonne formalisation pour l'implémentation de la sémantique. – Bien adaptée pour le développement d'outils. – Permet la description d'états et d'opérations.	– Non adaptée aux preuves et à l'analyse formelle. – S'appuie sur la sémantique d'une machine à états abstraite (ASM).

TABLE 3.1 – Comparaison des sémantiques

2 Approches et outils pour la définition de sémantiques et l'exécution de modèles

Dans cette section, nous présentons plusieurs approches qui permettent de définir des sémantiques de langages de modélisation et/ou fournissent les outils nécessaires pour l'exécution des modèles. Nous pressentons d'abord un ensemble de critères d'évaluation afin d'analyser et évaluer chaque approche. Ensuite, nous donnons la description de chaque approche. Enfin, on conclue par une synthèse des différentes évaluations.

2.1 Critères de comparaison

On rappelle que notre objectif principal est l'étude et la mise en œuvre d'un moteur d'exécution de modèles temps réel qui exploite les hypothèses sur la sémantique d'exécution des différents modèles d'une manière précise et cela à des niveaux d'abstraction élevés. La solution que nous allons proposer et développer doit répondre à quartes critères essentiels. Ces critères permettent d'analyser et de comparer les différentes approches retenues et de justifier les choix effectués dans cette thèse. L'ensemble de critères identifié est :

1. L'aspect flot de conception/outillage est important dans notre étude. Pour cela on veut connaitre l'effort et le coût de l'intégration de ces approches dans un flot de conception respectant les préconisations du MDA. Cela nécessite l'identification de la proximité des formalismes utilisés dans chaque approche par rapport aux préconisations du MDA pour mettre en œuvre ce type de flots.

   Comme nous l'avons décrit en section 1.2, un flot de conception qui adopte les préconisations du MDA doit respecter principalement les deux points suivants :

   - ≻ Utiliser le standard MOF [19] pour la méta-modélisation.
   - ≻ Utiliser le standard UML [18] pour la modélisation.

   Nous allons donc comparer, dans ce qui suit, les formalismes des différentes approches par rapport à UML et MOF.
2. Les applications temps réel embarquées, impliquent par nature, l'utilisation de différentes variantes sémantiques afin de capturer la spécificité de chaque domaine d'application. Ce critère d'évaluation positionne les approches retenues par rapport à leur flexibilité pour prendre en compte les particularités de différents domaines (notamment en liens avec l'utilisation de profils UML, impliquée par le premier critère d'évaluation).
3. La simulation d'un modèle nécessite une sémantique précise et bien définie. Ce troisième critère de comparaison est de s'assurer si les approches existantes s'appuient sur des fondations sémantiques précises et standardisées, non seulement pour expliciter les hypothèses relatives à la sémantique d'exécution supportée par le moteur

d'exécution, mais aussi pour pouvoir, à terme, rendre ces hypothèses disponibles pour toute projection sémantique requise par une activité d'analyse.

4. Le quatrième critère d'évaluation concerne la capacité des approches existantes de simuler le comportement d'applications temporisées et concurrentes, en lien avec les besoins des systèmes temps-réel embarqués, et ce à des niveaux d'abstractions relativement élevés. La simulation dans ce cas, où les systèmes sont par nature contraints en ressources et en temps est utilisée pour tester le comportement du système non seulement d'un point de vue fonctionnel, mais aussi d'un point de vue temporel et concurrentiel.

2.2 Kermeta

Kermeta [58] est une approche de méta-modélisation exécutable. C'est une extension de EMOF (Essential Meta-Object Facilities) permettant la description d'aspects comportementaux.

Kermeta est composé d'un package structurel et d'un package comportemental. Le premier package correspond aux éléments de EMOF. Le second package correspond à un ensemble de classes hiérarchiques qui représente les expressions impératives pour la description de la sémantique. En pratique, définir la sémantique d'exécution d'un langage revient d'abord à définir le méta-modèle et les opérations nécessaires pour l'exécution des modèles en utilisant le package structurel de Kermeta. Ensuite la sémantique d'exécution est décrite dans le corps des opérations en utilisant le langage d'action impératif et orienté objet défini par Kermeta.

À titre d'exemple, la figure 3.1 illustre un méta-modèle simple de machines à états finis (FSM), exprimé à l'aide du package structurel de Kermeta. Ce méta-modèle définit la syntaxe abstraite du langage. Le méta-modèle est enrichi par un ensemble d'opérations et d'attributs afin de capturer le caractère exécutable du langage. La sémantique d'exécution est décrite de manière opérationnelle.

La figure 3.2 montre le code de l'opération *run* de l'élément *FSM*. Cette opération est décrite à l'aide du langage d'action de Kermeta et capture de manière opérationnelle les règles de déclenchement de l'exécution d'une machine à états et de franchissement des transitions entre les différents états. L'imbrication des appels aux différentes opérations définies dans le méta-modèle fournit une description de la sémantique d'exécution du langage, et permet ainsi d'exécuter n'importe quel modèle conforme à ce méta-modèle.

2.2.1 Évaluation

- **Critère N°1**

Kermeta respect les préconisations du MDA pour la méta-modélisation. Le formalisme utilisé est conforme aux standards de l'OMG. Quant au niveau modélisation, Kermeta

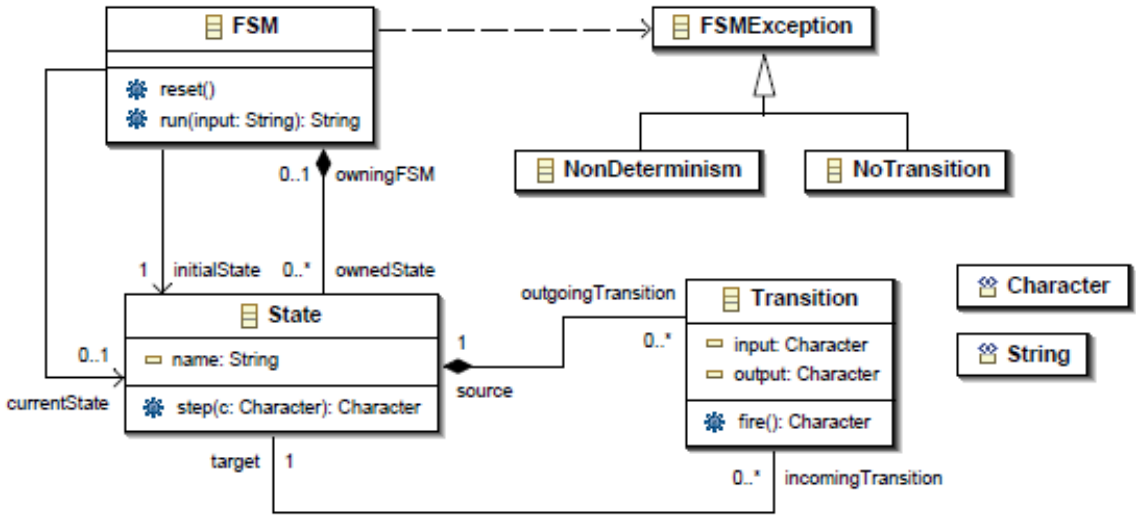


FIGURE 3.1 – Un méta-modèle simple pour les machines à états finis (Source : [58])

```
operation run(input : String) : String raises FSMException is do
    // reset if there is no current state
    if currentState == void then reset end
    // initialise result
    result := ""
    from var i : Integer init 0
    until input.size == i
    loop
      result.append( currentState.step( input.charAt(i) ).toString )
      i := i + 1
    end
end
```

FIGURE 3.2 – Code de l'opération *run* de l'élément *FSM* du méta-modèle FSM (Source : [58])

utilise les DSMLs obtenues pour la définition des modèles. Utiliser UML dans ce cas, revient à redéfinir le méta-modèle du langage UML en suivant la méthodologie utilisée dans Kermeta. L'intégration de cette approche dans un flot de conception utilisant UML, nécessite un effort considérable de développement.

➢ **Critère N°2**

Kermeta propose un bon cadre méthodologique pour définir la sémantique d'exécution d'un DSML. Cette approche est assez flexible pour capturer les particularités d'un domaine. Elle ne pose aucune contrainte pour la définition du DSML. C'est au développeur d'anticiper les particularités du langage dans la phase de définition des différents méta-modèles. Il doit ajouter les points d'extensions ainsi que les mécanismes nécessaires pour gérer une éventuelle variante syntaxique et sémantique. Quant à l'utilisation des profils UML par cette approche, elle reste limitée. L'adaptation

de l'utilisation des profils UML avec un DSML définie par Kermeta nécessite un effort considérable de développement (Cela reviens à redéfinir le méta-modèle ou un sous ensemble du langage UML et le profil UML dans Kermeta ainsi que le mécanisme d'application de ce profil).

≻ **Critère N°3**

La sémantique d'un DSML est spécifiée d'une manière opérationnelle au niveau des opérations du méta-modèle de la syntaxique abstraite. Les hypothèses relatives à la sémantique d'exécution pourraient être non seulement utilisé pour des fins d'exécutions mais permettent aussi de réaliser d'autres activités par exemple le model-checking [64].

≻ **Critère N°4**

Kermeta est un langage de méta-modélisation exécutable plutôt qu'un outil de simulation de modèles. C'est un framework structuré permettant d'exploiter la définition sémantique pour obtenir un environnement de simulation. La simulation des comportements concurrents et temporisés dans Kermeta, nécessite d'abord la prise en compte de ces aspects dans la définition du langage de modélisation. Cela revient à préciser comment les comportements concurrents ainsi que les contraintes temporelles seront représentés. Ensuite, il faut spécifier la sémantique d'exécution de chaque élément syntaxique du langage.

2.3 TopCased

TopCased [65] est un environnement d'IDM par les modèles pour le développement des systèmes logiciels et matériels embarqués. Il propose un ensemble d'outils ayant pour but de simplifier la définition de nouveaux langages de modélisation spécifique à un domaine (DSML). Ces outils s'appuient principalement sur les technologies Eclipse [66], EMF [67] et GMF [26] en offrant des générateurs d'éditeurs syntaxiques (textuels ou graphiques), des outils de validation statique (contraintes OCL) ainsi que des outils de vérification et de validation dynamique. Ce dernier point est réalisé par la simulation de modèles. Les DSML envisagés dans TopCased se focalisent essentiellement sur des modèles asynchrones ou synchrones à événements discrets.

TopCased fournit également une méthodologie pour définir une sémantique d'exécution d'un langage de modélisation. Cette méthodologie est résumée dans les étapes suivantes :

1. Définir la syntaxe abstraite du langage de modélisation sous la forme du quadruplet $MM = \langle MMs, MMd, MMe, MMt \rangle$ présenté sur la figure 3.3 où :

   ≻ MMs est le méta-modèle structurel ou statique du langage. Il définit les différents concepts du langage. C'est l'équivalent du méta-modèle classique d'un langage de modélisation.

➢ MMd est le méta-modèle dynamique. Il capture toutes les informations requises durant la simulation des modèles. Ces informations sont ajoutées au méta-modèle structurel sous forme d'attributs, de relations et de nouveaux éléments.

➢ MMe est le méta-modèle des événements. Il définit d'une part les événements qui orchestrent l'exécution des modèles, d'autre part, les événements produits par l'exécution des modèles.

➢ MMt est le méta-modèle de traces. Il spécifie les traces générées par la simulation des modèles.

2. La sémantique d'exécution du langage est spécifiée sous forme opérationnelle dans le moteur de simulation de TopCased.

Le moteur de simulation s'occupe de mettre à jour les informations dynamiques du modèle (MMd) en fonction des occurrences d'événements (MMe) définies dans un scénario de simulation. Il est composé d'un moteur générique et d'une définition de la sémantique pour chaque DSML. Le moteur générique implante le modèle de calcul à événements discrets. Le composant spécifique à chaque DSML implémente la sémantique opérationnelle en langage Java. Il définit comment les informations dynamiques évoluent en réaction à chacun des événements.

2.3.1 Évaluation

➢ **Critère N°1**

TopCased respect les préconisations du MDA pour la méta-modélisation. Le formalisme utilisé est conforme aux standards de l'OMG. Quant au niveau modélisation, TopCased utilise les DSMLs obtenues pour la définition des modèles. Utiliser UML dans ce cas, revient à redéfinir le méta-modèle du langage UML en suivant la méthodologie utilisée dans TopCased. L'intégration de cette approche dans un flot de conception utilisant UML, nécessite un effort considérable de développement.

➢ **Critère N°2**

L'approche définie dans TopCased est assez flexible pour capturer les particularités d'un domaine. Elle ne pose aucune contrainte pour la définition du DSML. C'est au développeur d'anticiper les particularités du langage dans la phase de définition des différents méta-modèles. Il doit ajouter les points d'extensions ainsi que les mécanismes nécessaires pour gérer une éventuelle variante syntaxique et sémantique. Quant à l'utilisation des profils UML par cette approche, elle reste limitée. L'adaptation de l'utilisation des profils UML avec un DSML définie par TopCased nécessite un effort considérable de développement (Cela reviens à redéfinir le méta-modèle ou un sous

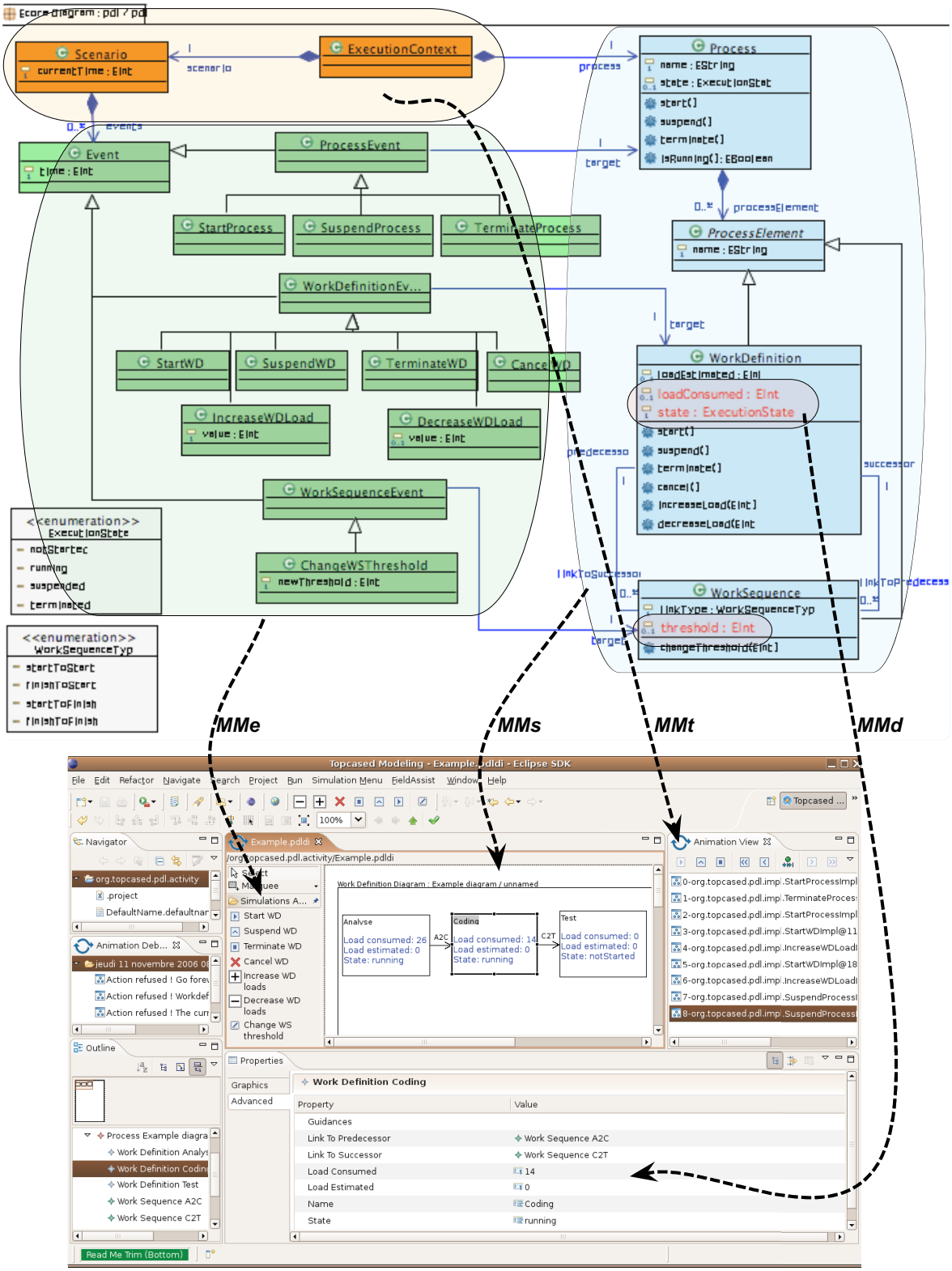


FIGURE 3.3 – L'impact des méta-modèles identifiés sur l'interface utilisateur de TopCased (Source : [68]

ensemble du langage UML et le profil UML dans TopCased ainsi que le mécanisme d'application de ce profil).

➢ **Critère N°3**

TopCased définit la sémantique d'un DSML d'une manière opérationnelle. Contrairement à Kermeta, la sémantique n'est pas spécifiée au niveau des opérations du méta-modèle mais elle est spécifiée au niveau d'un module séparé du moteur d'exécution. Ce module est consacré à la capture de la sémantique du DSML. La sémantique est exprimée en langage Java.

➢ **Critère N°4**

Dans TopCased, le cadre méthodologique pour la définition d'un DSML et la spécification de sa sémantique s'appuie sur des langages existants. La démarche à suivre pour supporter les exécutions concurrentes et temporisées est équivalente à celle expliquée dans la deuxième évaluation de Kermeta.

2.4 Semantic Units

Cette approche [69] utilise le principe de la *Semantic Unit* ou *unité sémantique* pour donner une sémantique opérationnelle à un DSML. La syntaxe abstraite du DSML est définie à l'aide de l'outil *GME* [70] sous forme d'un méta-modèle décrit en UML. L'unité sémantique se compose de :

➢ Un modèle de données abstrait.

➢ Un ensemble de règles opérationnelles (opérations) qui manipulent les éléments identifiés dans le modèle de données abstrait.

Les deux composantes de l'unité sémantique sont décrites en *AsmL* [71] (Abstract State Machine Langage). Ce langage a une sémantique formelle. Il permet de spécifier des machines d'état abstraites [72].

La sémantique du DSML est spécifiée par un mapping entre une unité sémantique et le méta-modèle du langage. Ce mapping est décrit dans l'outil *GReAT* [73]. C'est un outil de transformation de modèle. Il permet de définir l'ensemble de règles permettant d'associer chaque élément du méta-modèle du langage à un élément dans le modèle de données abstrait de l'unité sémantique. Les différentes étapes de cette approche sont présentées dans la figure 3.4.

L'exemple suivant explique l'utilisation d'une unité sémantique pour donner une sémantique à des machines à états finis. La figure 3.5 illustre le méta-modèle de la syntaxe abstraite du langage. Le modèle de données abstrait ainsi que l'ensemble des opérations de l'unité sémantique associée à ce langage sont présentés par la figure 3.6. Elle montre la description en *AsmL* des éléments *FSM*, *event*, *state* et *transition* qui correspondent respectivement à une machine à états finis, un événement, un état et une transition. La

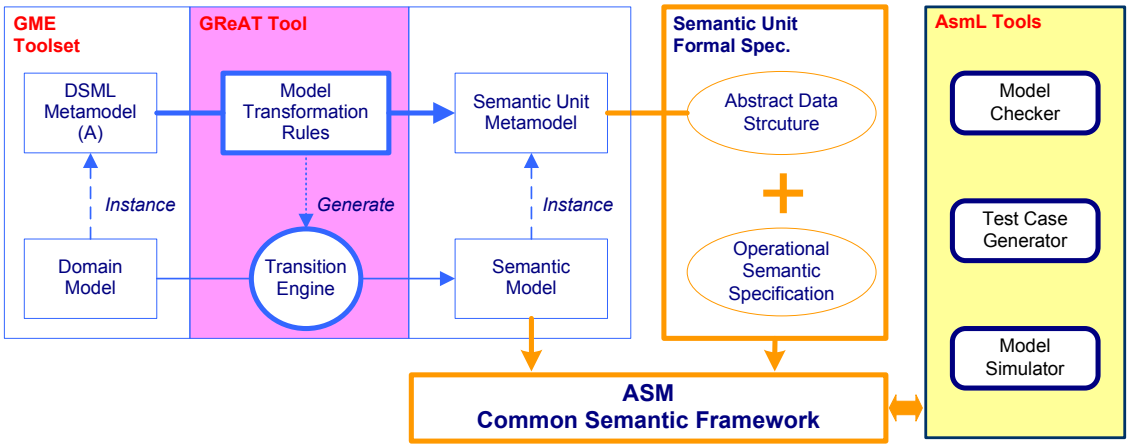


FIGURE 3.4 – Vue d'ensemble de l'approche basée sur les unités sémantiques (Source : [69])

figure 3.7 présente le code définit en *AsmL* de l'opération *fsmReact*. Cette opération permet de calculer la réaction de la machine à états à un événement.

La définition des règles de mapping entre les éléments du méta-modèle du langage (figure 3.5) et les éléments du modèle de données abstrait de l'unité sémantique (figure 3.6) nécessite la réécriture du modèle de données abstrait sous forme d'un méta-modèle. Ce méta-modèle est décrit dans le même langage que celui utilisé pour décrire la syntaxe abstraite. La figure 3.8 montre le modèle de données abstrait décrit par un méta-modèle. La figure 3.9 présente l'une des règles de mapping décrite par l'outil *GReAT*.

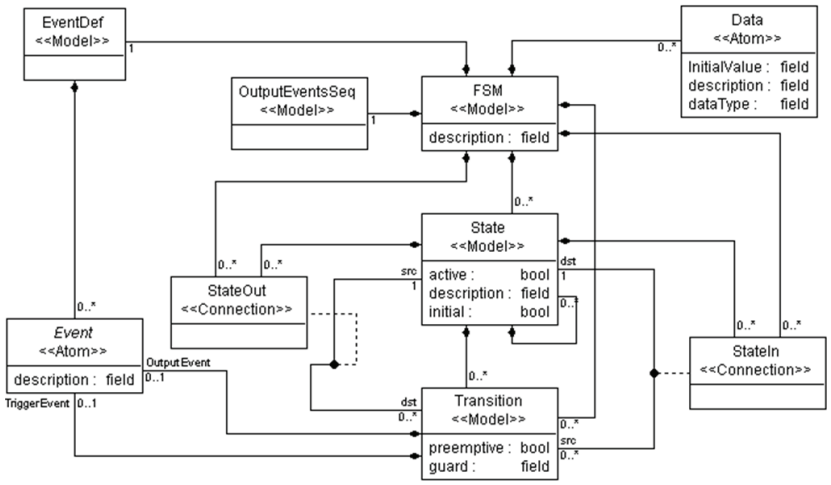


FIGURE 3.5 – Méta-modèle de la syntaxe abstraite des machines à états finis (Source : [69])

2.4.1 Évaluation

➢ **Critère N°1**

Semantic Units respect les préconisations du MDA pour la méta-modélisation. Le formalisme utilisé est conforme aux standards de l'OMG. Quant au niveau

```
structure Event
class FSM
  var  outputEvetns as Seq of Event
  var  initialState  as State
  var  children      as Set of State

class State
  var active              as Boolean = false
  var initial             as Boolean
  var initialState        as State?
  var parentState         as State?
  var slaves              as Set of State
  var outTransitions      as Set of Transition

class Transition
  var guard               as Boolean
  var preemptive          as Boolean
  var triggerEvent        as Event?
  var outputEvent         as Event?
  var srcState            as State
  var dstState            as State
```

FIGURE 3.6 – Modèle abstrait de données de l'unité sémantique (Source : [69])

```
fsmReact (fsm as FSM, e as Event) =
  step let s as State = getCurrentState (fsm, e)
  step let pt as Transition? = getPreemptiveTransition (fsm, s, e)
  step
    if pt <> null then doTransition (fsm, s, pt)
    else
      step
        if isHierarchicalState (s) then invokeSlaves (fsm, s, e)
        let npt as Transition? = getNonpreemptiveTransition (fsm, s, e)
      step
        if npt <> null then doTransition (fsm, s, npt)
```

FIGURE 3.7 – Code de l'opération *fsmReact* (Source : [69])

modélisation, Semantic Units utilise les DSMLs obtenues pour la définition des modèles. Utiliser UML dans ce cas, revient à redéfinir le méta-modèle du langage UML en suivant la méthodologie utilisée dans Semantic Units. L'intégration de cette approche dans un flot de conception utilisant UML, nécessite un effort considérable de développement.

➢ **Critère N°2**

L'approche Semantic Units est assez flexible pour capturer les particularités d'un domaine. Elle ne pose aucune contrainte pour la définition du DSML. C'est au développeur d'anticiper les particularités du langage dans la phase de définition des différents méta-modèles. Il doit ajouter les points d'extensions ainsi que les mécanismes nécessaires pour gérer une éventuelle variante syntaxique et sémantique. Quant à l'utilisation des profils UML par cette approche, elle reste limitée. L'adaptation de l'utilisation des profils UML avec un DSML définie par Semantic Units nécessite un

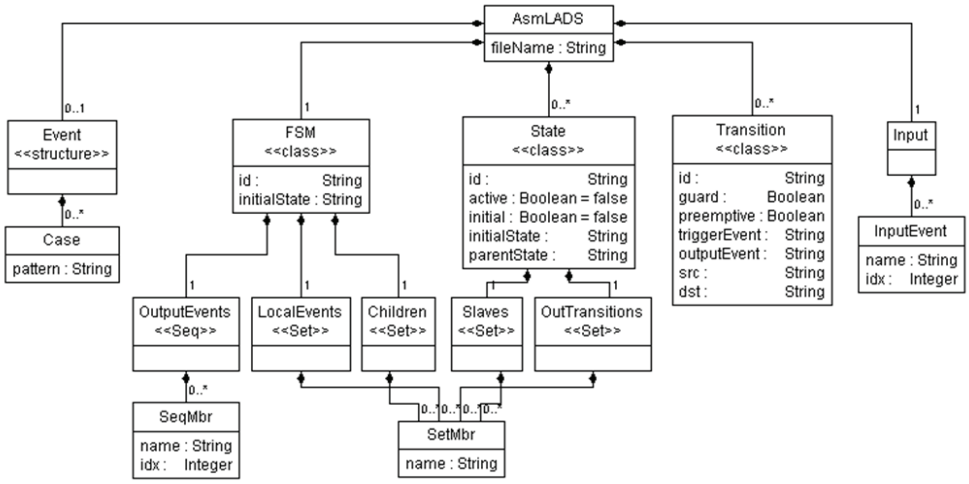


FIGURE 3.8 – Méta-modèle de modèle de données abstrait de l'unité sémantique (Source : [69])

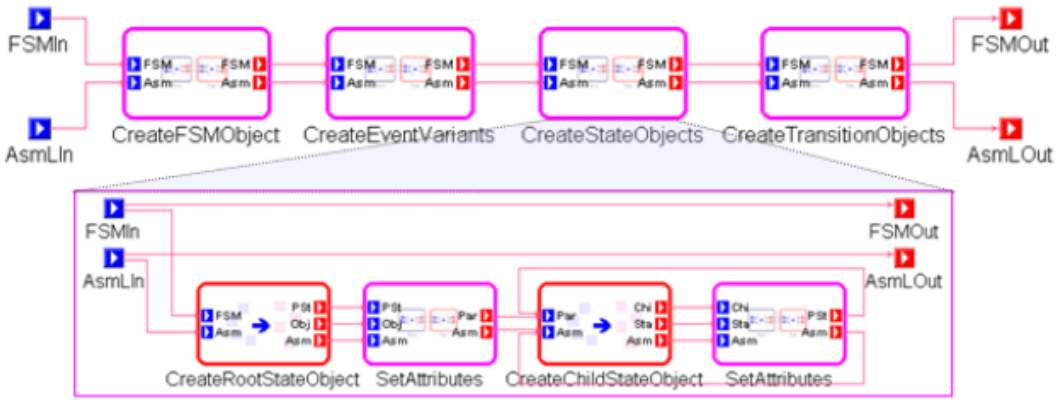


FIGURE 3.9 – Régle de mapping entre le méta-modèle de la syntaxe abstraite et le méta-modèle du modèle de données abstraits (Source : [69])

effort considérable de développement (Cela reviens à redéfinir le méta-modèle ou un sous ensemble du langage UML et le profil UML dans avec Semantic Units ainsi que le mécanisme d'application de ce profil).

➢ **Critère N°3**

Semantic Units repose sur la notion de machines à états abstraites pour la spécification de la sémantique. La sémantique est spécifiée d'une manière opérationnelle en utilisant le langage *ASML* [71] afin de simuler le comportement des modèles. Le principe des machines à états abstraites fournit un cadre formel standardisé qui permet la définition de la sémantique d'exécution de n'importe quel langage. D'ailleurs, la sémantique de plusieurs langages est définie en utilisant ce principe, tel que le langage SDL-2000 [74] et le langage VHDL [75].

➢ **Critère N°4**

Semantic Units est un framework structuré permettant d'exploiter la définition

sémantique pour obtenir un environnement de simulation. La spécification de la sémantique d'un DSML s'appuie sur cadre formel et standardisé. La démarche à suivre pour supporter les exécutions concurrentes et temporisées est équivalente à celle expliquée dans la deuxième évaluation de Kermeta.

2.5 MagicDraw/Cameo

MagicDraw [76] est un environnement de modélisation UML. Cameo [77] est un plugin MagicDraw. Il permet de simuler et vérifier les modèles UML et SysML [78] capturés dans MagicDraw. L'environnement MagicDraw/Cameo n'impose aucune méthodologie particulière pour capturer les modèles des systèmes. L'utilisateur a le libre choix de décrire son système en fonction de sa démarche de développement. La seule contrainte à prendre en compte est que les éléments qui peuvent être exécutés doivent être pris en charge par les moteurs de simulation de Cameo. Seulement les activités UML, les machines à états et les diagrammes paramétriques SysML sont supportés pour les exécutions. La figure 3.10 donne un aperçu de l'environnement MagicDraw/Cameo. Les caractéristiques principales de Cameo sont les suivantes :

- Une infrastructure de simulation générique qui permet de contrôler et déboguer les simulations, animer les modèles.
- Une API ouverte [79] permettant de connecter plusieurs moteurs d'exécution.
- Un moteur d'exécution des activités UML basé sur une implémentation du standard OMG fUML. Les concepts des activités supportés pour la simulation se limitent aux éléments retenus dans fUML.
- Un moteur d'exécution des machines à états basé sur le standard W3C SCXML [80]. Ce standard fournit un environnement d'exécution générique basé sur les statecharts de Harel [81]. Cameo export les machine à états UML vers ce standard pour réaliser les simulations. Seulement un sous ensemble des machines états est supporté par ce moteur d'exécution.
- Un moteur d'exécution paramétrique pour l'exécution et le calcul des modèles mathématiques des systèmes. Le système sur lequel la simulation paramétrique peut être réalisée doit être modélisé par SysML. Les expressions mathématiques et logiques définis dans les modèles SysML sont résolus par un solveur mathématique interne. L'utilisation d'un solveur externe est possible via l'API ouverte de l'infrastructure générique de simulation.

2.5.1 Évaluation

- **Critère N°1**

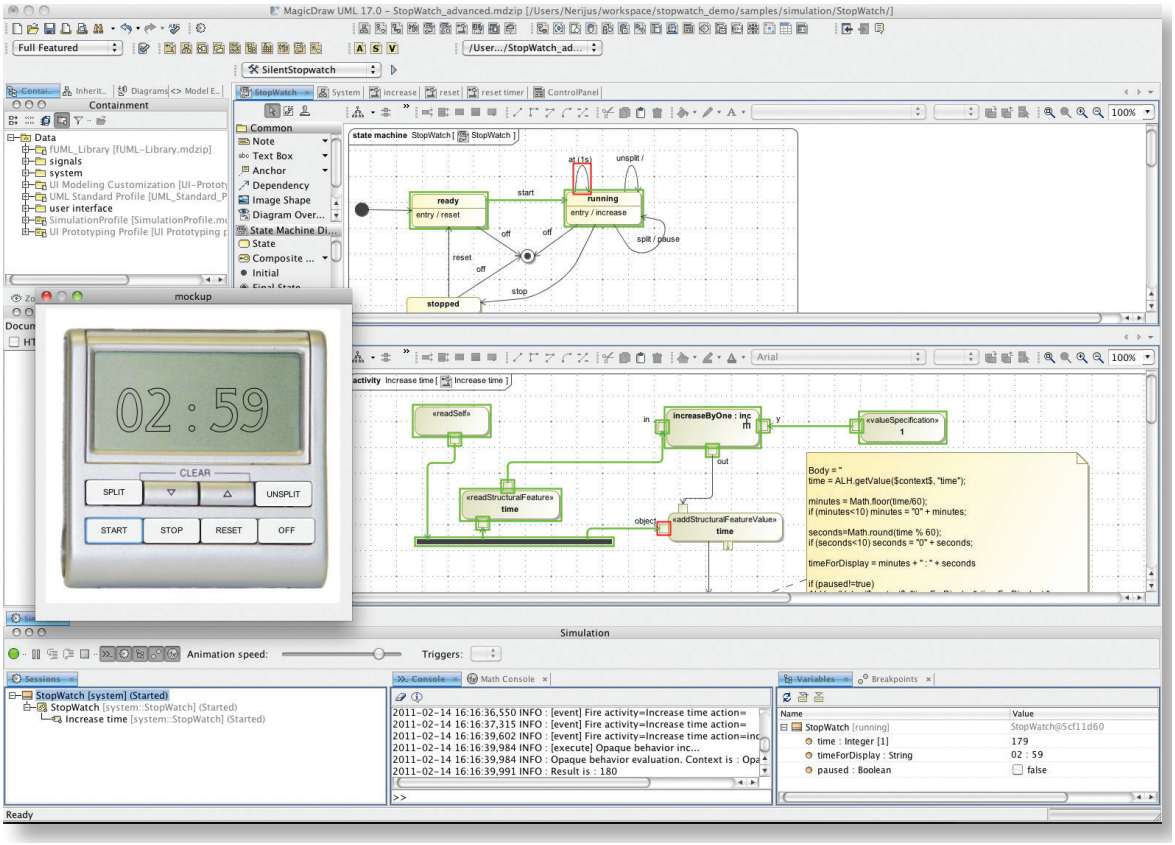


FIGURE 3.10 – Capture d'écran de l'environnement MagicDraw/Cameo

MagicDraw/Cameo est un environnement qui respecte les préconisations du MDA en offrant le moyen (l'API ouverte) pour son extension ou son intégration dans un autre flot de conception.

➢ **Critère N°2**

MagicDraw/Cameo est un outil commercial fermé. La sémantique du sous-ensemble d'UML retenue dans cet outil est figée. Il ne fournit aucun mécanisme qui permet de paramétrer la sémantique existante ou d'ajouter une nouvelle variante sémantique. En revanche, MagicDraw/Cameo autorise l'utilisation des profils UML pour l'adaptation des modèles à des domaines spécifiques, mais la sémantique impliquée par l'application des profils n'est pas prise en compte par les moteurs de simulation.

➢ **Critère N°3**

MagicDraw/Cameo utilise les activités et les machines à états pour décrire des aspects comportementaux de l'application modélisée pour la simulation. Le cadre sémantique de cet outil repose sur la sémantique donnée dans la norme fUML pour les activités et le standard W3C SCXML pour les machines à états.

➢ **Critère N°4**

MagicDraw/Cameo permet de modéliser la concurrence par le diagramme d'états-

transitions et le diagramme d'activités. Concernant la notion du temps, les événements temporisés (Time events) sont utilisés pour modéliser les déclenchements après un certain laps de temps. Les simulations sont réalisées par le moteur d'exécution de fUML et de W3C. Les résultats des simulations sont enregistrés en format XML.

2.6 iUML

iUML [82] est un environnement de modélisation et de simulation permettant la vérification des comportements modélisés par la simulation. iUML fournit aussi une méthodologie de modélisation pour capturer les modèles des systèmes. La figure 3.11 récapitule les différentes étapes de la démarche de modélisation.

En premier lieu, le système est partitionné en plusieurs sujets d'étude ou domaines où chaque domaine est modélisé par un package UML. Par la suite, chaque modèle de domaine est construit en trois étapes. La première étape consiste à élaborer un modèle statique ou structurel. Il permet de capturer les données requises par les calculs et les comportements en décrivant les entités conceptuelles du domaine en termes de classes, attributs et associations. La deuxième étape définit un modèle dynamique ou comportemental. Il s'agit d'une part de capturer les comportements asynchrones par des machines à états, représentées concrètement par des diagrammes d'états-transition d'UML. D'autre part, des opérations sont ajoutées aux classes afin de répondre aux appels synchrones. Durant la troisième étape, le langage d'action ASL (Action Specification Langage) [83] est utilisé pour décrire impérativement la sémantique d'exécution des opérations et des états.

Une fois que le modèle du système est complet, la définition d'un ensemble d'éléments supplémentaires est nécessaire pour pouvoir exécuter ce modèle. Ces éléments représentent essentiellement l'environnement d'exécution, c'est-à-dire l'ensemble d'objets et de signaux initiaux et les scénarios de tests. En effet, les exécutions de modèles dans iUML sont basées sur le modèle de calcul à événements discrets où l'exécution est avancée par étapes de consommation des signaux.

En outre, l'environnement iUML possède une architecture ouverte permettant d'ajouter un langage externe comme C/C++ au corps des opérations définie en ASL. Cela permet de connecter par exemple un modèle à une interface graphique ou un environnement de simulation externe.

2.6.1 Évaluation

➢ **Critère N°1**

iUML est un outil qui respect les préconisations du MDA mais reste fermé à l'extension ou à l'intégration dans un autre flot de conception.

➢ **Critère N°2**

iUML est un outil commercial fermé. La sémantique du sous-ensemble d'UML retenue

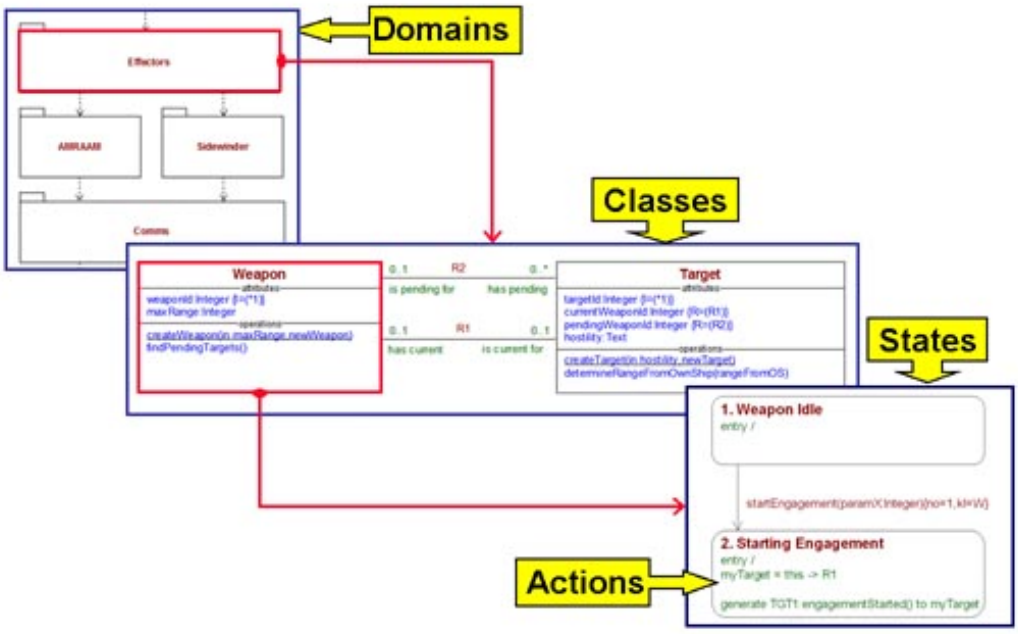


FIGURE 3.11 – Méthodologie de modélisation dans iUML

dans cet outil est figée. Il ne fournit aucun mécanisme qui permet de paramétrer la sémantique existante ou d'ajouter une nouvelle variante sémantique. De plus, il ne permet pas l'utilisation des profils UML pour l'adaptation des modèles à des domaines spécifiques.

➢ **Critère N°3**

iUML utilise le langage d'action *ASL* [83] pour décrire des aspects comportementaux de l'application modélisée pour la simulation. Le cadre sémantique de cet outil repose sur la sémantique du langage d'action ASL. Une partie de la sémantique est imposée dans le moteur d'exécution du simulateur, elle est basée sur le modèle de calcul à événements discrets.

➢ **Critère N°4**

iUML permet de modéliser la concurrence par les diagrammes d'états-transitions. Concernant les contraintes temporelles, elles sont représentées par la notion de Timer du langage d'action ASL [83]. Les simulations sont réalisées par le moteur d'exécution de iUML. Le résultat des simulations illustre les différentes exécutions concurrentes ainsi que les dates d'occurrence de chaque signal et opération.

2.7 Ptolemy

Ptolemy [84] est une approche qui s'intéresse à la modélisation de l'hétérogénéité, à la simulation et la conception des systèmes embarqués temps réel concurrents. C'est une approche basée sur la notion des acteurs communicants. L'acteur représente l'élément de

base dans un système. C'est une entité concurrente qui communique via une interface composée de ports.

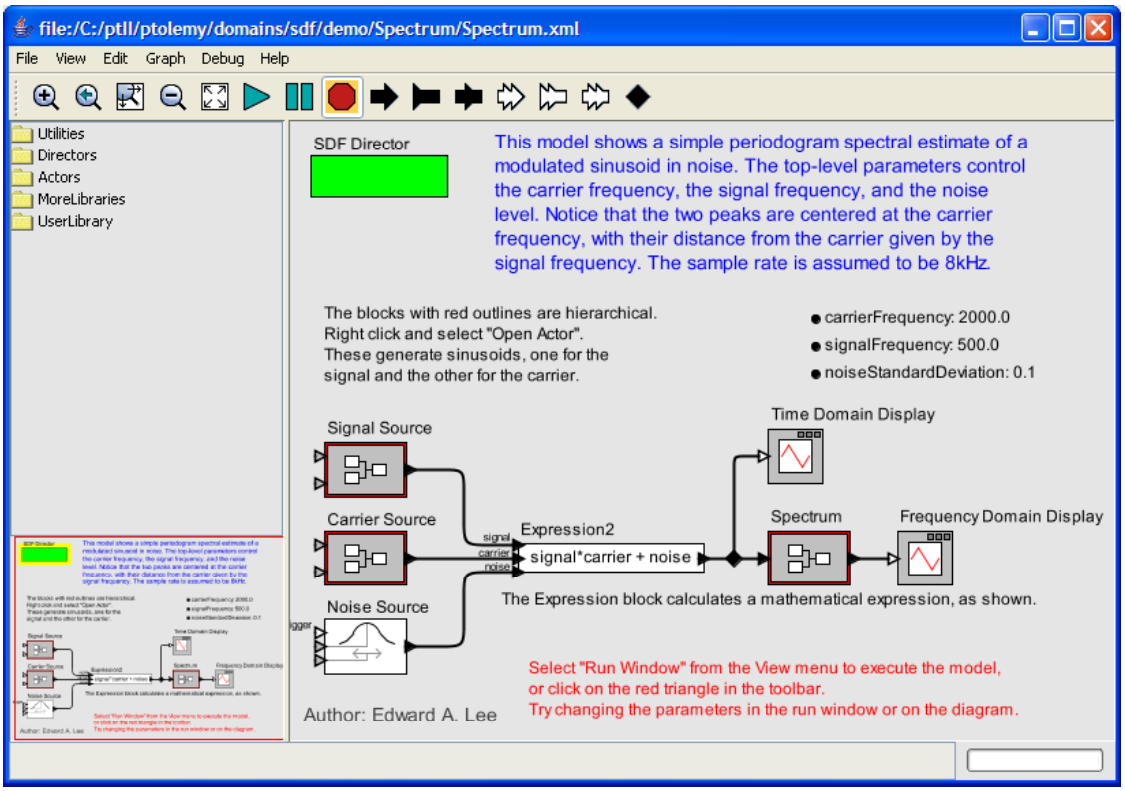


FIGURE 3.12 – Exemple de modèle capturé par Ptolemy

La structure d'un modèle est capturée en utilisant une syntaxe abstraite très simple à base de blocs qui représentent les acteurs, de ports et de relations. Chaque modèle est composé d'un ensemble d'acteurs qui possèdent des interfaces de communication. Chaque interface est composée d'un ensemble de ports. Les interactions entre les différents acteurs sont exprimées par des relations entre les ports des acteurs.

Ptolemy s'appuie sur la notion des modèles de calcul pour spécifier la sémantique des modèles. Pour cela, il associe un domaine à chaque modèle pour définir la sémantique donnée à une syntaxe. Le domaine définit les règles permettant d'exécuter un modèle en fonction d'un modèle de calcul particulier. Ces règles donnent la définition détaillée du modèle de calcul utilisé dans laquelle des informations telles que le type de communication et le mode de synchronisation entre acteurs sont spécifiées. L'ensemble des règles d'un modèle de calcul est implémenté par un directeur qui est responsable des simulations des modèles. La figure 3.12 présente un un exemple de modèle auquel un directeur du domaine SDF est associé.

Ptolemy implémente une large variété de modèles de calcul qui adressent les concepts de concurrence et de temps dans les systèmes de différentes manières. On trouve le modèle de calcul à flots de données synchrone (SDF), le modèle de calcul à événements discrets

(DE), le modèle de calcul des réseaux de processus (PN), le modèle de calcul des processus séquentiels communiquant par rendez-vous (CSP), etc. L'architecture de Ptolemy, fondée sur le principe de directeur, est en effet assez flexible pour prendre en compte différents modèles de calcul, c'est-à-dire prendre en compte plusieurs sémantiques d'exécution, et permettre leur combinaison pour construire des modèles hétérogènes.

2.7.1 Évaluation

➢ **Critère N°1**

Le formalisme utilisé dans Ptolemy pour la définition des modèles est assez loin des préconisations du MDA. L'intégration de cette approche dans un flot de conception basé sur UML nécessite un effort de développement considérable en matière de passerelles/transformation de modèles.

➢ **Critère N°2**

Ptolemy est une approche orientée composants. La syntaxe abstraite utilisée est très simple. Cette syntaxe n'est pas conforme au méta-modèle UML ce qui limite l'utilisation des profils UML. D'autre part, cette approche offre une grande flexibilité pour capturer les particularités sémantiques d'un domaine en se basant sur le principe de directeur.

➢ **Critère N°3**

Ptolemy se base sur la notion des modèles de calcul pour la spécification de la sémantique dont la finalité est la simulation et l'exécution des modèles. Les modèles de calcul sont définies d'une manière opérationnelle en code Java. Cette approche ne repose sur aucun support formel pour la définition de la sémantique.

➢ **Critère N°4**

Ptolemy support la simulation des comportements concurrents et temporosés à travers une large variété de modèles de calcul. Le parallélisme potentiel est capturé au niveau des diagrammes de blocs. Cependant le support des exécutions temporisé est réalisé à un niveau d'abstraction relativement bas (dans le code du modèle de calcul).

2.8 ModHel'X

ModHel'X [85] est un framework pour la modélisation des systèmes hétérogènes. Comme Ptolemy, il adopte une approche orientée composants. Un système est donc décomposé en éléments réalisant une ou plusieurs fonctions. La fonctionnalité globale du système est assurée par la coopération des différents éléments qui le composent. Ces éléments sont modélisés par un ensemble de blocs connectés par des arcs unidirectionnels. On distingue deux types de blocs : les blocs atomiques et les blocs composites. Un bloc atomique représente les composants de base du système. Un bloc composite est un bloc qui

encapsule d'autres blocs interconnectés les uns avec les autres. Les entrées et les sorties de chaque bloc sont modélisées par des ports. Les informations échangées entre les différents blocs sont représentées par des jetons.

ModHel'X utilise la notion de modèle de calcul pour donner une sémantique d'exécution à la structure d'un système. Le modèle de calcul fournit les règles d'exécution du modèle. Il définit la manière dont les différents blocs seront orchestrés afin d'obtenir une exécution valide.

ModHel'X propose un moteur de simulation générique pour exécuter le comportement d'un système. Ce moteur de simulation se base sur le principe d'observation (snapshot) pour interpréter la structure d'un modèle en fonction du modèle de calcul associé. Une exécution revient à réaliser une série d'observations du modèle. Afin de calculer chaque observation, l'algorithme de simulation, présenté dans la figure 3.13, prend en compte les entrées du modèle et construit les sorties en fonction de l'état courant du modèle.

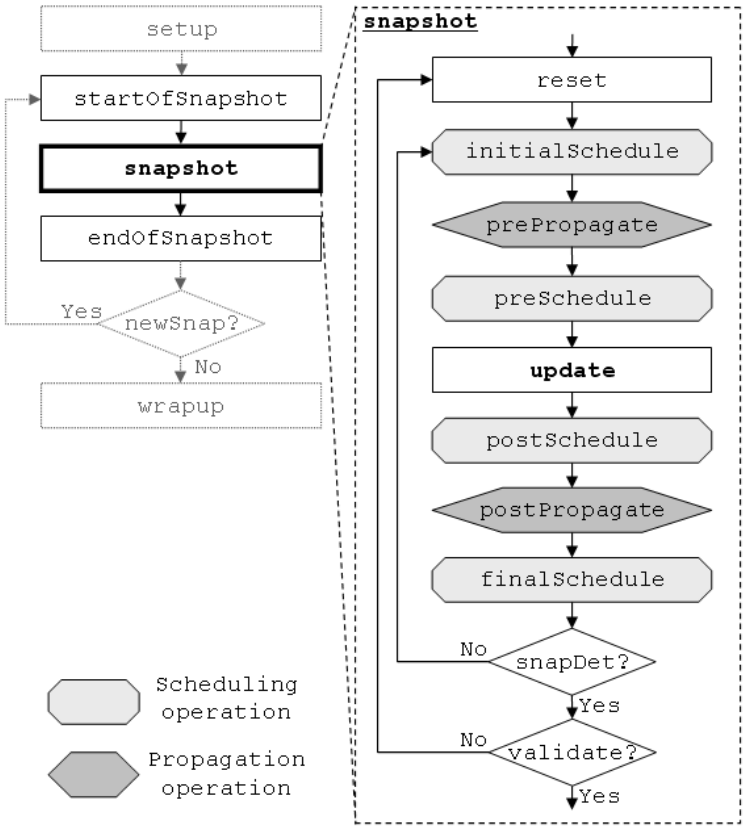


FIGURE 3.13 – L'algorithme d'exécution général de ModHel'X

Cet algorithme se décompose en trois étapes :

- Choisir le bloc à observer.
- Observer le comportement du bloc choisit en fonction de ses entrées.
- Propager les données (jetons) obtenues selon les relations (arcs) dans la structure du modèle.

Ces étapes s'appuient sur des opérations primitives qui peuvent être raffinées pour chaque modèle de calcul. L'implémentation de ces opérations définit la sémantique du modèle de calcul.

ModHel'X dispose pour l'instant de moins de modèles de calcul que Ptolemy, mais son architecture et son moteur d'exécution générique comportent des mécanismes qui permettent de mieux contrôler les interactions entre différents modèles de calcul au sein d'un modèle hétérogène.

2.8.1 Évaluation

➢ **Critère N°1**

Le formalisme utilisé dans ModHel'X pour la définition des modèles est assez loin des préconisations du MDA. L'intégration de cette approche dans un flot de conception basé sur UML nécessite un effort de développement considérable en matière de passerelles/transformation de modèles.

➢ **Critère N°2**

ModHel'X est une approche orientée composants. La syntaxe abstraite utilisée est très simple. Cette syntaxe n'est pas conforme au méta-modèle UML ce qui limite l'utilisation des profils UML. ModHel'X propose un moteur d'exécution générique qui permet à travers ses opérations primitives de capturer différentes variantes sémantiques.

➢ **Critère N°3**

ModHel'X se base sur la notion des modèles de calcul pour la spécification de la sémantique dont la finalité est la simulation et l'exécution des modèles. Les modèles de calcul sont définies d'une manière opérationnelle en code Java. Cette approche ne repose sur aucun support formel pour la définition de la sémantique.

➢ **Critère N°4**

ModHel'X support la simulation des comportements concurrents et temporosés à travers une large variété de modèles de calcul. Le parallélisme potentiel est capturé au niveau des diagrammes de blocs. Cependant le support des exécutions temporisé est réalisé à un niveau d'abstraction relativement bas (dans le code du modèle de calcul).

2.9 Synthèse

La figure 3.14 presente un tableau qui synthétise la comparaison des différentes approches présentées par rapport aux critères que nous avons identifiés.

Les approches telles que Kermeta et TopCased proposent un bon cadre méthodologique pour la définition de la sémantique d'un DSML pour obtenir un environnement de simulation, sans être fermé à des éventuelles évolutions et extensions, mais négligent

	Critères de comparaison			
Outils	***Critère N°1***	***Critère N°2***	***Critère N°3***	***Critère N°4***
Kermeta	XX	XX	XX	XX
TopCased	XX	XX	XX	XX
Semantic Units	XX	XX	XXX	XX
MagicDraw/Cameo	XXX	XX	XXX	XX
iUML	X	X	XX	XXX
ModHel'X	X	XX	XX	XXX
Ptolemy	X	XX	XX	XXX

XXX : satisfait
XX : partiellement satisfait
X : non satisfait

FIGURE 3.14 – Tableau récapitulatif de la comparaison

l'aspect formel et standardisé de la sémantique. Semantic Units appartient à la famille de TopCased et Kermeta sauf qu'elle repose sur un cadre formel pour la définition de la sémantique. iUML et MagicDraw/Cameo sont des bons environnements de simulation de modèles UML qui respectent les préconisations du MDA, mais restent très fermés quant au support de différentes variantes sémantique. L'intégration dans un flot de conception MDA de MagicDraw/Cameo est plus facile par rapport à iUML. Ptolemy et ModHel'X se caractérisent par un excellent environnement de simulation et une grande flexibilité en matière de support de plusieurs sémantiques d'exécutions, mais reste loin des préconisations du MDA. Ce qui rend difficile l'intégration de ces deux approches dans un flot de conception MDA.

Dans la section suivante, nous allons présenter le standard OMG « Semantics of a Foundational Subset for Executable UML Models ». Ce standard nous semble une bonne base de départ pour atteindre nos objectifs car il satisfait les quatre critères d'évaluation présentés dans la section 2.1. Cela revient à :

- fUML propose un moteur d'exécution pour la simulation des modèles UML.

- fUML offre un cadre formel et standardisé pour la définition de la sémantique tout en respectant les préconisations du MDA.

3 fUML

Cette section introduit fUML et ses concepts fondamentaux. Elle présente une analyse en identifiant les limites de fUML par rapport aux besoins du domaine des applications temps réel.

3.1 Introduction

Le *"Semantics of a Foundational Subset of Excecutable UML Models"* [86] est une nouvelle norme de l'OMG qui définit une sémantique d'exécution pour un sous-ensemble du méta-modèle d'UML, appelé *"Foundational UML (fUML)"*. Ce sous-ensemble correspond à la syntaxe abstraite de la norme. Il inclut un nombre réduit de concepts UML afin de faciliter la définition d'une sémantique claire, précise et non ambigüe. La sémantique est formalisée sous forme d'un modèle d'exécution spécifié dans un style opérationnel. Ce modèle d'exécution traite les concepts encadrés en rouge de la figure 3.15. Ces concepts englobent les fondements structurels de modélisation, les comportements inter et intra objets, les activités et les actions.

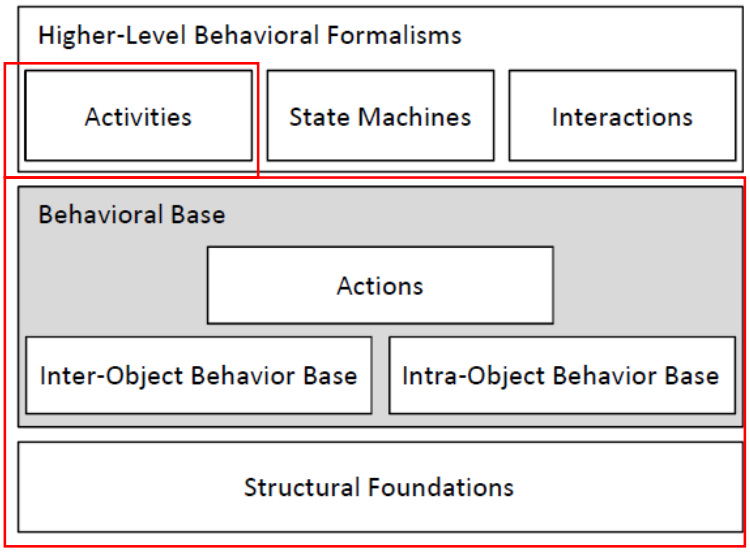


FIGURE 3.15 – Couches sémantiques d'UML

3.2 La syntaxe : Concepts de modélisation

La syntaxe abstraite de fUML est un sous-ensemble minimal du méta-modèle UML. Pour une partie des éléments retenus des contraintes additionnelles sont définies. Le tableau 3.2 montre les concepts de modélisation retenus. Nous trouvons alors les notions de classes et d'associations pour les aspects de modélisation structurelle. Concernant la modélisation du comportement fUML ne garde que les activités et les actions.

Les packages UML	Inclus dans fUML ?
Modélisation structurelle	
Classes	**Oui**
Components	Non
Composite Structures	Non
Deployments	Non
Modélisation comportementale	
Actions	**Oui**
Activities	**Oui**
Common Behaviors	**Oui**
Interactions	Non
State Machines	Non
Use Cases	Non

TABLE 3.2 – Les packages d'UML retenus dans fUML

3.3 La sémantique : Le modèle d'exécution de fUML

Le modèle d'exécution est une spécification opérationnelle en Java de la sémantique de fUML. Il spécifie notamment la sémantique d'exécution des concepts de modélisation inclus dans le sous-ensemble fUML.

Le modèle d'exécution est lui-même un modèle défini à l'aide d'un sous-ensemble nommé bUML (base UML). Cette définition circulaire est rompue par une description axiomatique de premier ordre de la sémantique (voir la section 10 de [86]).

Le modèle d'exécution est structuré en packages, symétriques au packages structurant le modèle de la partie syntaxique. Pour chaque package de la syntaxe abstraite, un package est introduit dans le modèle d'exécution à l'exception d'un package,nommé Loci, qui n'a pas de correspondance avec les éléments syntaxiques. Ce package définit le moteur et l'environnement d'exécution des modèles fUML.

La définition du modèle d'exécution de fUML est basée sur le patron de conception *visiteur* [87]. Son principe est d'associer des comportements à une hiérarchie de classes existante sans modifier la structure. Dans le contexte de fUML, il permet d'attacher un comportement aux concepts de modélisation sans changer le méta-modèle de la syntaxe abstraite (clause 7 de [86]). Concrètement, pour chaque méta-classe de fUML, un comportement est donc défini via une classe visiteur dans le modèle d'exécution. Chaque nouvelle classe a une association unidirectionnelle avec la méta-classe de l'élément syntaxique et définit les opérations nécessaires pour capturer la sémantique de cet élément. Toutes les classes visiteur héritent directement de la classe *SemanticVisitor* du modèle d'exécution. Nous distinguons trois types de classes visiteurs, illustrés dans la figure 3.16.

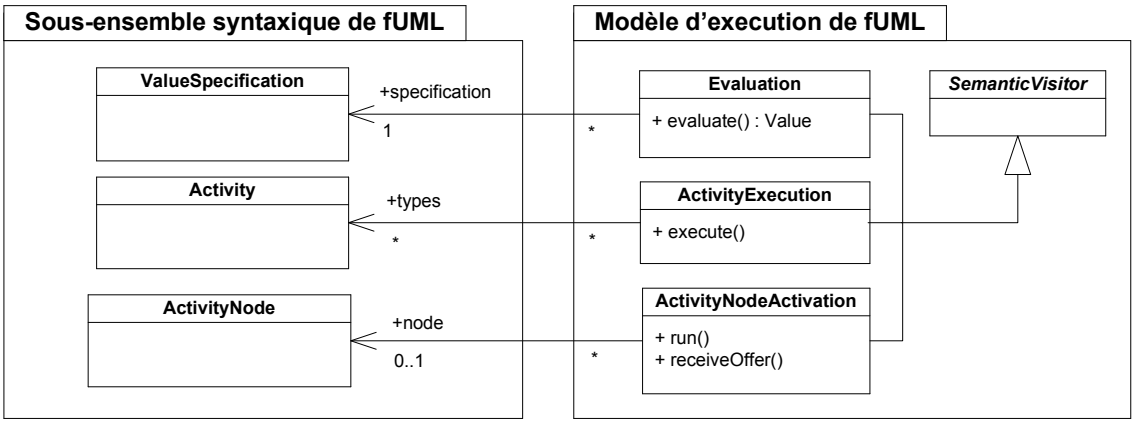


FIGURE 3.16 – Extrait du modèle d'exécution de fUML montrant les types de classes visiteurs

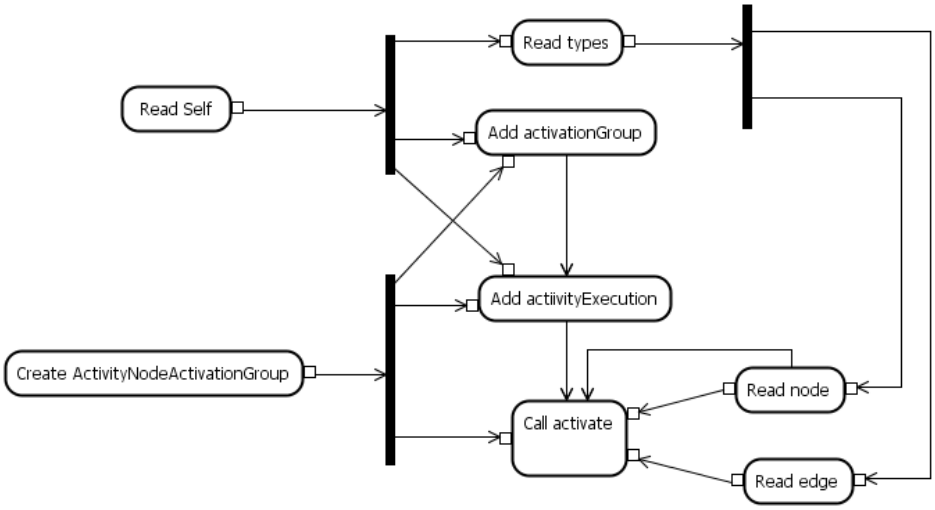
➢ **Les classes visiteurs de type *Execution* :** ce type de classe est utilisé pour décrire le comportement de chaque sous-classe concrète de Behavior, inclus dans fUML. Nous trouvons la classe de base *ActivityExecution* qui correspond aux activités. Il existe notamment deux autres classes *OpaqueBehaviorExecution* et *FunctionBehaviorExecution* qui correspondent respectivement à OpaqueBehavior et FunctionBehvaior de la syntaxe abstraite.

➢ **Les classes visiteurs de type *Activation* :** ce type de visiteur spécifie la sémantique des différents nœuds d'activité. Par exemple la classe visiteur *SendSignalActionActivation* correspond à l'action SendSignalAction.

➢ **Les classes visiteurs de type *Evaluation* :** ce type de visiteur est utilisé pour spécifier comment une sous-classe concrète de ValueSpecification est évaluée. Par exemple la classe visiteur *LiteralBooleanEvaluation* décrit comment une valeur booléenne (instance de la classe LiteralBoolean) est évaluée.

Chaque classe visiteur du modèle d'exécution possède des opérations qui capturent la sémantique d'exécution. Le comportement de chacune de ces opérations est décrit sous la forme d'activités fUML. Pour faciliter la lisibilité de ces activités, le langage Java est utilisé comme syntaxe concrète à la place de diagrammes d'activités. Cette utilisation de Java suit des règles syntaxiques strictes, qui permettent une projection non-ambigûe vers une représentation sous forme d'activités. Ces règles de projection sont décrites dans [86],annexe A.

Par exemple, la partie supérieure de la figure 3.17 montre un diagramme d'activité partiel de l'opération *execute()* de la classe *ActivityExecution* du modèle d'exécution. La partie inférieure de la figure 3.17 montre sa description en Java en respectant le passage de Java vers les activités définit dans la norme.



Activity activity = (fUML.Syntax.Activity) (this.types.getValue(0));
ActivityNodeActivationGroup group = new ActivityNodeActivationGroup () ;
this.activationGroup = group;
group.activityExecution = this;

FIGURE 3.17 – Diagramme d'activité partiel d'une opération du modèle d'exécution et sa représentation équivalente en Java

3.3.1 Points de variation sémantique

Les points de variation sémantique dans fUML sont traités en utilisant le patron de conception de stratégie [87], représenté par la classe *SemanticStrategy* dans le modèle d'exécution. Son principe est de définir une classe abstraite stratégie par point de variation sémantique possédant une opération abstraite. Ensuite plusieurs sous-classes concrètes de la classe abstraite sont définies pour implémenter différents comportements, chaque sous-classe concrète définissant une variante sémantique pour le point de variation correspondant. fUML traite deux points de variation sémantique dans son modèle d'exécution présentés dans la figure 3.18.

Le premier point adresse le choix d'événements parmi la liste d'événements reçus par un objet. La classe stratégie abstraite correspondante est *GetNextEventStrategy*. La sémantique est fixée dans la classe concrète *FIFOGetNextEventStrategy* qui fournit une implémentation d'une politique FIFO traitant les événements reçus par ordre d'arrivée.

Le deuxième point concerne l'appel polymorphique d'opération, c'est-à-dire la détermination de quelle méthode utiliser pour un appel d'opération. Cela est lié à la surcharge des opérations dans la même hiérarchie de classes. Une opération peut avoir plusieurs comportement en fonction de son ordre dans la hiérarchie de classes. L'opération *dispatch* de classe *Object* spécifie la sémantique de la détermination de l'opération à appeler dans une hiérarchie de classes. La classe stratégie abstraite correspondante est

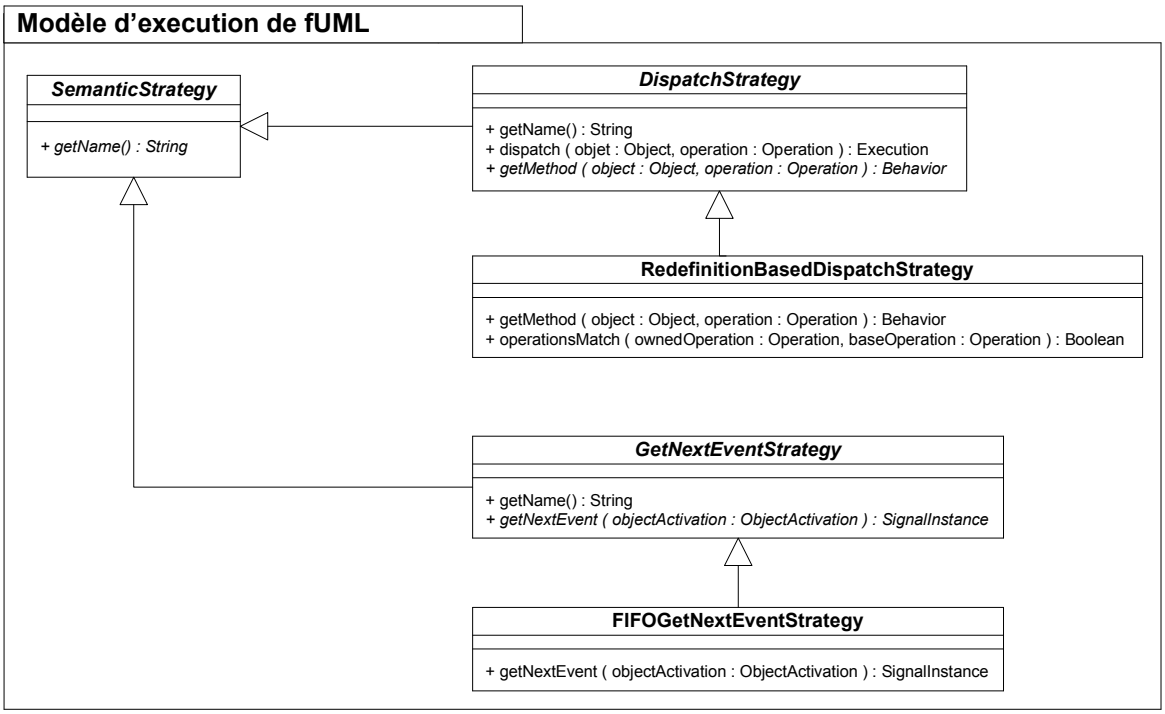


FIGURE 3.18 – Les deux points de variation sémantique du modèle d'exécution de fUML

DispatchStrategy. Le sémantique par défaut dans le modèle d'exécution est fixé par la classe concrète *RedefinitionBasedDispatchStrategy*.

3.4 Le moteur d'exécution et son environnement

Les éléments du moteur d'exécution de fUML et de son environnement sont regroupés dans le package Loci représenté par la figure 3.19. Il contient trois classes essentielles : *Locus*, *Executor* et *ExecutionFactory*. Ces classes décrivent une machine virtuelle ou (un interpréteur) de modèles fUML. La classe *Locus* représente l'unité de calcul ou le processeur virtuel qui exécute les modèles. Cette classe englobe tous les objets créés durant une exécution. La classe *Executor* représente une interface donnant accès au moteur d'exécution. Elle fournit les services nécessaires pour le lancement des exécutions. Quant à la classe *ExecutionFactory*, elle est utilisée pour créer les instances des différentes classes visiteurs du modèle d'exécution.

En effet, afin de pouvoir commencer une exécution, un environnement d'exécution initial est nécessaire. Cet environnement est un ensemble d'objets collaboratifs, composé de :

- Une seule instance de la classe *Locus*.
- Une seule instance de la classe *Executor*.
- Une seule instance de la classe *ExecutionFacotry*.
- Une seule instance par chaque sous-classe concrète de type stratégie.

Les étapes d'initialisation du moteur d'exécution en vue de démarrer une exécution

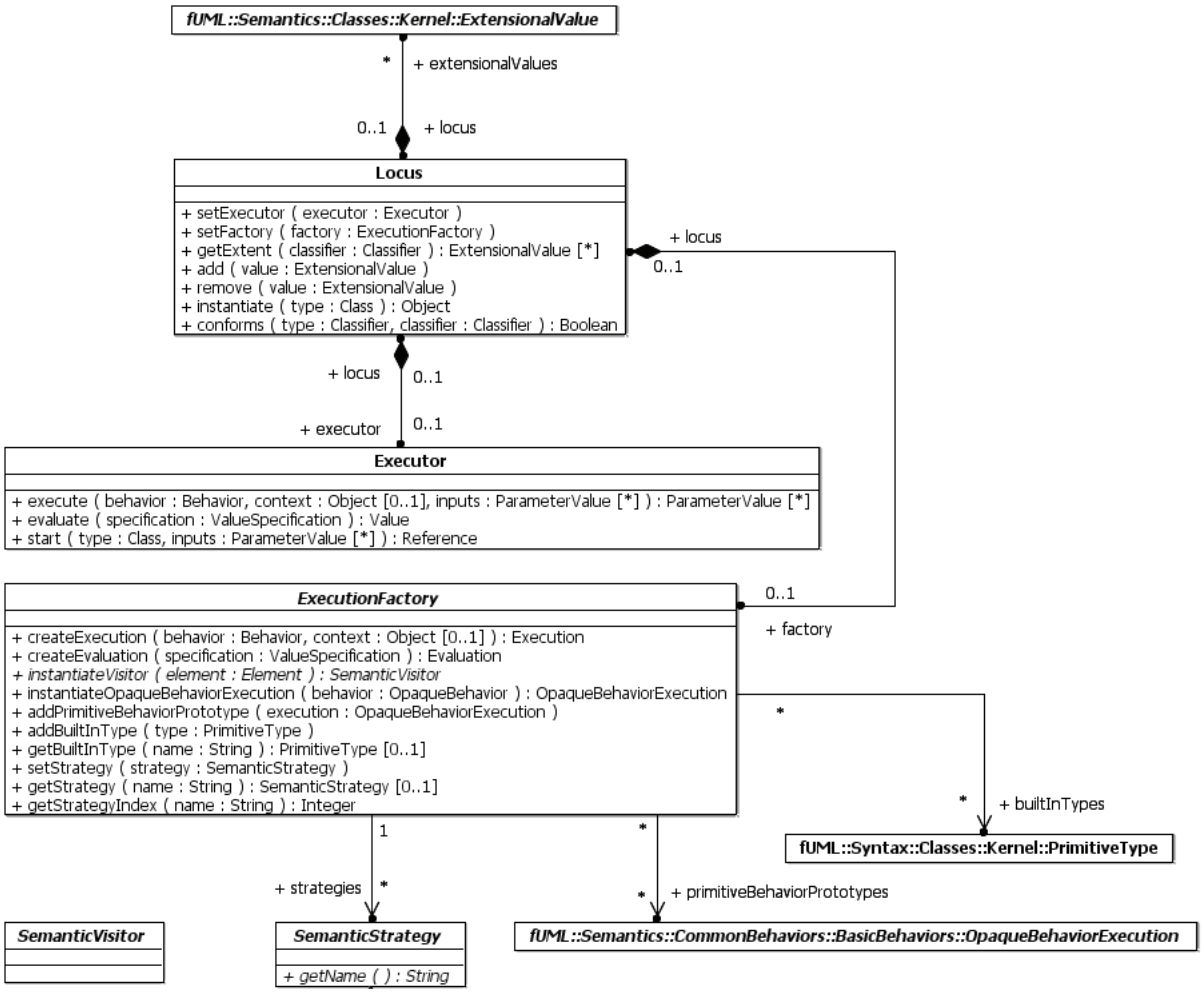


FIGURE 3.19 – Le package Loci du modèle d'exécution de fUML

sont illustrées par le diagramme de séquence de la figure 3.20. Le lancement du moteur d'exécution commence par la classe *main* FUML. Les différents étapes sont :

- ≻ Créer et configurer l'environnement d'exécution en instanciant les classes nécessaires.
- ≻ Initialiser l'exécution par l'appel d'opération *execute()* de la classe *ExecutionEnvironement* en passant en paramètre le l'activité à exécuter.
- ≻ Cette activité est passée par la suite en paramètre à l'opération *execute()* de la classe *Executor*, en ajoutant des paramètres initiaux.
- ≻ Créer la classe visiteur correspondant à l'activité passée en paramètre par l'appel de l'opération *createExecution()* de la classe *ExecutionFactory*.

Toutes ces étapes représentent seulement une initialisation de l'environnement pour pourvoir lancer une exécution. L'exécution d'une activité démarre effectivement en appelant l'opération *execute()* de la classe *ActivityExecution*.

Pour montrer l'intérêt de fUML, [88] a proposé une implémentation, réalisée en langage

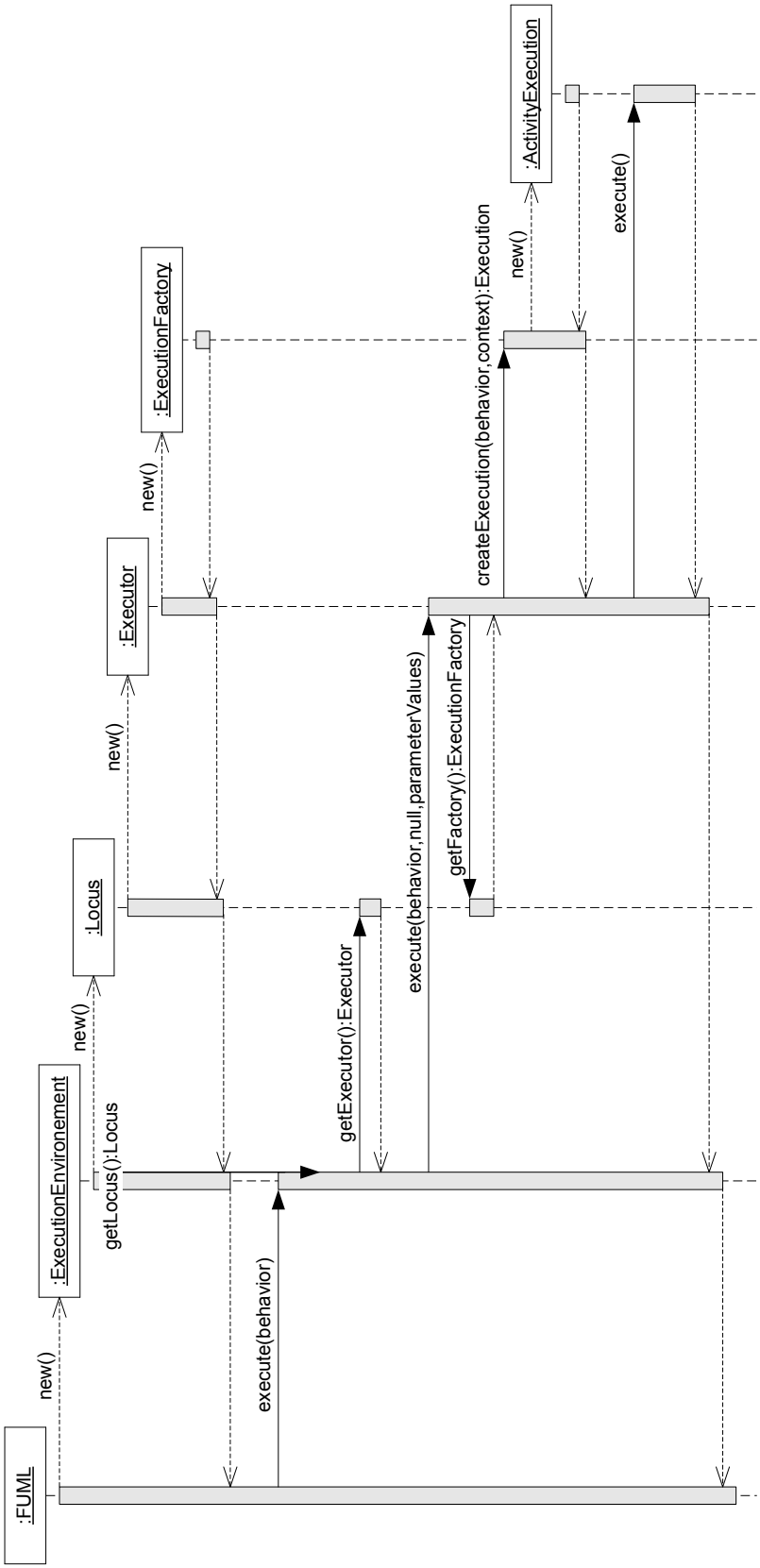


FIGURE 3.20 – Diagrammes de séquence de l'initialisation du moteur d'exécution de fUML

Java, tout en respectant les spécifications de la norme. Ainsi, elle fournit exactement les mêmes classes de l'environnement d'exécution décrit précédemment. De plus, toutes les descriptions des classes du modèle d'exécution restent inchangées. En outre, cette implémentation propose en complément un parseur de modèle. Ce parseur permet de parcourir des modèles fUML, stockés en format XMI afin de les exécuter.

3.5 L'exécution des activités

Une fois le moteur d'exécution initialisé, l'exécution de l'activité est déclenchée par un appel de l'opération *execute()* sur l'instance de la classe visiteur du modèle d'exécution. Cette opération crée d'abord les différentes instances des classes visiteurs qui correspondent à chacun des nœuds inclus dans l'activité. La partie gauche de la figure 3.22 illustre les éléments de la syntaxe abstraite du diagramme d'activité de la figure 3.21. La partie droite de la figure 3.22 montre toutes les instances du modèle d'exécution créées pour accomplir l'exécution de l'activité.

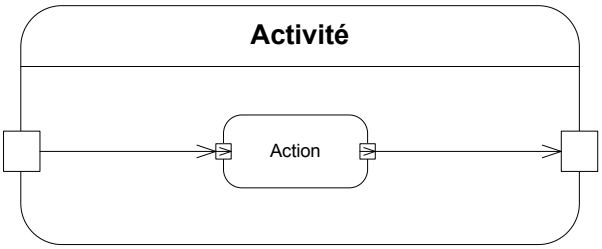


FIGURE 3.21 – Diagramme d'activité d'une simple activité

Concrètement, l'exécution se déroule par plusieurs appels d'opérations entre ces différentes instances. Ces appels respectent les étapes suivantes :

- Fournir les valeurs des paramètres d'entrée de l'activité. Le moteur d'exécution vérifie la présence de toutes les valeurs nécessaires aux nœuds d'entrée.
- Identifier les nœuds activés dans l'activité, à partir desquels l'exécution commence. Les nœuds activés sont les nœuds initiaux, les nœuds de paramètres d'entrées et les actions qui n'ont pas d'arcs entrants.
- L'envoie d'un jeton de contrôle à chaque nœud activé.
- L'exécution du comportement du nœud sélectionné. Dans le cas des nœuds activés, la sélection est effectué selon l'ordre de création dans le modèle. Dans le ces des autres nœuds la sélection se fait en fonction des relations de données et de contrôles. En effet, l'exécution d'un nœud suit les étapes suivantes :

  ≻ A : Vérifier si le nœud sélectionné est prêt à être exécuté. Cela consiste à vérifier si tous les jetons nécessaires à l'exécution du comportement sont disponibles.

  ≻ B : Si tous les prérequis sont disponibles, le nœud consomme ses jetons.

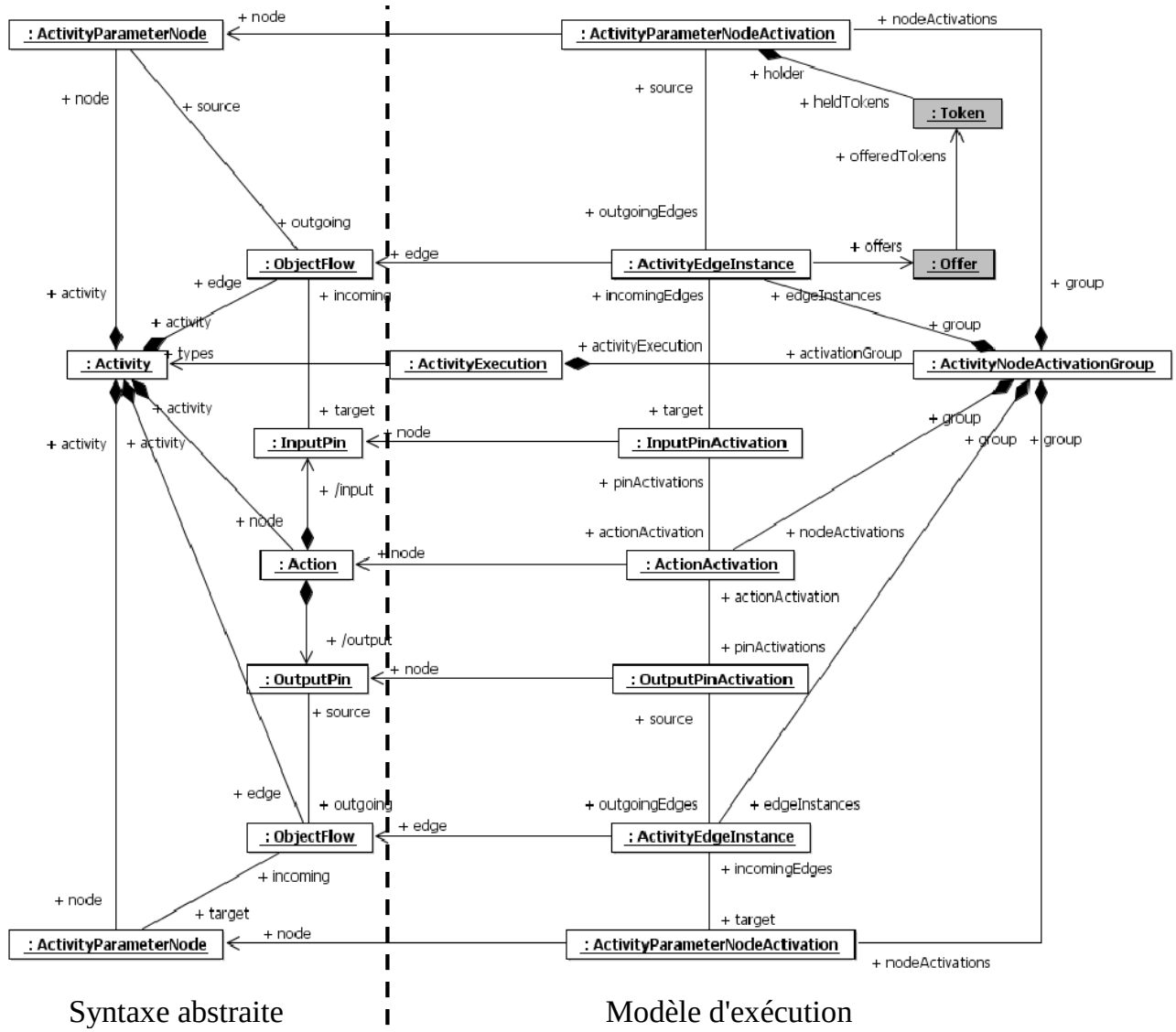


FIGURE 3.22 – La syntaxe abstraite et le modèle d'exécution d'une simple activité

Dans le cas contraire, un autre nœud sera sélectionné et la phase A sera répétée.

➢ C : Déclencher le comportement propre au nœud. Les résultats de l'exécution sont produits sous forme de jetons.

➢ D : Propager les jetons de sortie aux nœuds successeurs.

➢ E : Exécuter un nœud successeur puis réitérer les phases de A à D.

– Produire le résultat final de l'exécution de l'activité. Quand tous les nœuds sont exécutés, l'activité termine son exécution et les résultats sont placés dans les nœuds de paramètres de sorties.

3.6 Analyse de fUML

Cette section présente une analyse de fUML et de son modèle d'exécution. Cette analyse montre les limites que nous avons identifiées par rapport aux besoins du domaine du temps réel. D'abord, nous illustrons les limitations concernant les exécutions concurrentes. Ensuite, nous nous intéressons à la notion du temps dans fUML. La dernière partie est consacrée aux limites relatives à la spécialisation sémantique par les profils.

3.6.1 Les exécutions concurrentes

fUML inclut les concepts nécessaires pour modéliser des systèmes concurrents. Il s'agit de l'objet actif [20] [21] et les communications asynchrones via les signaux (Signal, SendSignalAction, SignalEvent). La concurrence est exprimée aussi en termes de flots d'actions dans une activité. C'est-à-dire, tous les flots capturés syntaxiquement en parallèle peuvent s'exécuter d'une manière concurrente à condition qu'il n'y ait pas de dépendance de données ou de contrôle entre ces différents flots de l'activité. Cependant, même si la capture syntaxique de la concurrence est bien maitrisée, la définition sémantique dans le modèle d'exécution est complètement séquentielle. En conséquence, l'exécution d'un modèle fUML concurrent produit une trace d'exécution séquentielle mais valide. Il s'agit d'une trace d'exécution parmi plusieurs exécutions parallèles potentielles. Cela est très contraignant dans le cadre d'une application temps réel, où l'observation des comportements concurrents est essentielle, comme nous l'avons montré dans le chapitre 1, section 2.

L'exemple suivant explique le problème de concurrence dans fUML. La figure 3.23 montre le modèle applicatif d'une activité contenant deux flots. Le premier flot est représenté par les actions A, B, C. Le deuxième flot est représenté par les actions D, E. Comme il n'existe aucune dépendance entre les deux flots, leur exécution devrait être concurrente. En revanche, en suivant le modèle d'exécution de fUML, l'interprétation de cette activité est purement séquentielle.

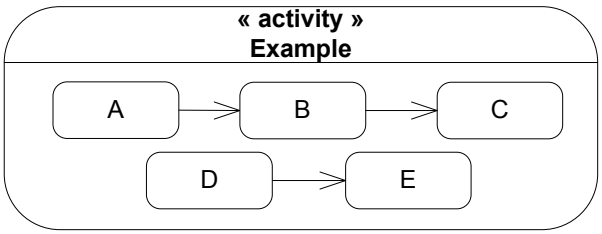


FIGURE 3.23 – Diagramme d'activité composé de deux flots parallèles

Le diagramme de séquence de la figure 3.24 illustre la trace d'exécution séquentielle de l'activité. Les lignes de vie représentent les instances des classes du modèle d'exécution. L'interaction entre les lignes de vie correspond à l'exécution de l'activité par le moteur d'exécution de fUML. L'instance de la classe *AvtivityExecution* représente l'exécution globale de l'activité. Les autres instances de classe représentent l'exécution des nœuds de l'activité.

Premièrement, l'exécution est déclenchée par l'appel de l'opération *execute()* de la classe *ActivityExecution*. Ensuite, la classe *ActivityExecution* appelle l'opération *receiveOffer()* sur un nœud activé de l'activité, dans ce cas c'est l'action A. Il est à noter que l'action D pourrait être choisie en premier, sauf que le choix est effectué selon l'ordre de création dans le modèle.

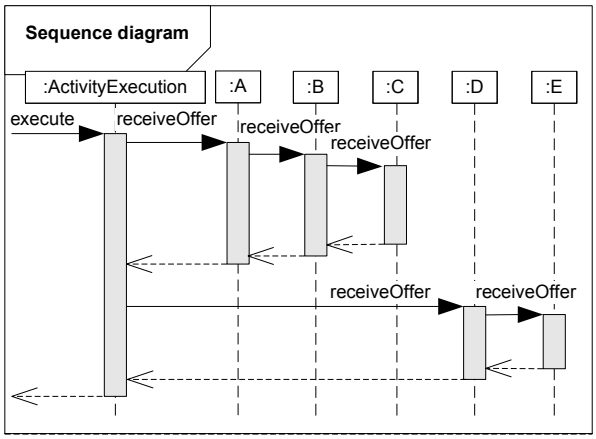


FIGURE 3.24 – Diagramme de séquence de l'exécution de deux flots parallèle (illustrés dans la figure 3.23 par le moteur d'exécution de fUML

Une fois que l'action A est lancée, l'exécution se déroule séquentiellement en propageant les appels des opérations jusqu'à la fin de l'exécution du premier flot (ABC). Quand l'exécution arrive à la fin du premier flot, l'instance d'*ActivityExecution* reprend l'exécution de l'activité et démarre le deuxième flot (DE) en appelant l'opération *receivOffer()* sur l'action D, qui propage séquentiellement à son tour les appels d'opérations. Il est important de noter qu'un seul thread réalise toute l'exécution de l'activité et produit une trace d'exécution qui reste valide mais ne représente qu'un déroulement possible parmi plusieurs. En d'autres termes, l'architecture du modèle d'exécution de fUML ne contient pas une entité pour contrôler et sélectionner l'action à exécuter. Ce mécanisme est enfoui dans l'implémentation de chaque classe visiteur du modèle d'exécution. D'après le diagramme de séquence nous constatons qu'on ne peut pas suspendre l'exécution après l'action B et lancer l'action D à cause de la propagation séquentielle des appels d'opérations. Cela peut cacher des erreurs et limite par conséquent les qualités des tests de simulation.

3.6.2 Le temps

fUML ne propose aucune définition de la sémantique du temps (voir le sous section 2.3 de [86]). D'une part fUML n'inclut aucun élément syntaxique qui permet de capturer la notion de temps. D'autre part, le modèle d'exécution ne fournit aucun mécanisme permettant la manipulation des exécutions de nature temporisée et toutes les propriétés sous-jacentes telles que échéance, période, etc.

3.6.3 La prise en charge des Profils

Le moyen standard pour étendre UML au domaine du temps réel et de l'embarqué est l'utilisation de profils (MARTE [34], UML-RT [89], RT-UML/Rhapsody [47], ACCORD [90],UML4ESL [91]). Dans le contexte de fUML, le sous-ensemble syntaxique n'inclut aucun concept lié aux profils. Cela implique que, le modèle d'exécution de fUML ignore les stéréotypes lors de l'exécution des modèles. En d'autres termes, l'architecture actuelle du modèle d'exécution ne fournit aucun moyen qui permette de personnaliser et prendre en compte la spécialisation sémantique impliquée par l'utilisation d'un profil.

3.7 Évaluation

➢ **Critère N°1**

fUML est une norme de l'OMG qui respect les préconisations du MDA. fUML est définit de manière à pouvoir l'intégrer facilement dans les flots de conception MDA.

➢ **Critère N°2**

fUML n'inclut aucun mécanisme permettant de capturer les particularités d'un domaine par l'application des profils. De plus, le modèle d'exécution de fUML ne fournit aucun mécanisme qui permet de paramétrer la sémantique existante ou d'ajouter une nouvelle variante sémantique.

➢ **Critère N°3**

fUML formalise la sémantique sous forme d'un modèle d'exécution. La sémantique est spécifiée d'une manière opérationnelle en utilisant le langage Java .

➢ **Critère N°4**

fUML définit moteur d'exécution qui permet de simuler les modèles conformes au sous ensemble fUML. Ce moteur d'exécution ne permet pas de simuler les comportements concurrents et temporisés.

En résumé, fUML fournit un bon cadre pour la définition de la sémantique et la simulation de modèles. Elle respect complètement les préconisations du MDA. C'est un standard ouvert aux extensions et aux intégrations dans des flots de conceptions MDA. Malgré les limitations concernant la capacité de simuler des comportements concurrents et temporisés et la prise en charge de nouvelles variantes sémantiques, l'effort de résoudre ces limitations est relativement faible par rapport aux approches et outils présentés dans la section 2 de ce chapitre. Pour cela, fUML reste une bonne base pour réaliser notre objectif.

CHAPITRE 4

Contribution

L'objectif de ce chapitre est de proposer des solutions pour surmonter les limitations de fUML identifiées dans le chapitre précédent. Ces solutions nous permettront de paramétrer le modèle d'exécution afin d'exécuter et de simuler des modèles annotés par des profils pour le domaine du temps réel.

Pour cela, ce chapitre est structuré de la manière suivante :

- Dans un premier temps nous traitant d'abord les aspects de concurrence et d'ordonnancement. Cette partie présente une extension du modèle d'exécution de fUML permettant d'adopter et de contrôler plusieurs politiques d'ordonnancements.
- La deuxième partie présente une deuxième extension du modèle d'exécution par laquelle la notion de temps est introduite.
- La troisième partie de ce chapitre, décrit une extension de fUML et de son modèle d'exécution, pour permettre l'application des profils au niveau modèle et le paramétrage du modèle d'exécution afin de spécifier les sémantiques d'exécution spécifiques au profil appliqué.
- La dernière section illustre l'utilisation du modèle d'exécution étendue sur un cas d'étude : Nous allons utiliser un sous profil de MARTE utilisé notamment dans la méthodologie de modélisation Optimum [93]. Les modèles annotés par ce profil seront simulés pour générer des traces d'exécution représentatives des caractéristiques temps réel.

1 L'ordonnancement dans fUML

Nous avons montré dans le chapitre de l'état de l'art (chapitre 3) que toute exécution suivant le modèle d'exécution de fUML est séquentielle. Nous avons souligné ainsi l'absence d'une entité explicite responsable de l'ordonnancement et de l'exécution des actions. La résolution de ce problème nécessite des extensions du modèle d'exécution de fUML.

Notre idée repose sur deux points. Dans un premier temps, nous proposons de rompre la propagation séquentielle des appels d'opération en donnant le moyen de contrôler le début et la fin d'exécution de chaque action dans une activité. Cela signifie qu'une fois l'exécution d'une action terminée, on donne la main à une entité qui décidera en fonction d'une politique d'ordonnancement, quelle est la prochaine action à exécuter tout en respectant les dépendances de données et de contrôle. Dans un second temps, nous introduisons un ordonnanceur dans le modèle d'exécution afin de contrôler les exécutions des différentes actions.

La figure 4.1 illustre l'ordonnanceur introduit dans le modèle d'exécution. L'ordonnanceur est représenté par la classe *Scheduler*. Il manipule une liste *d'ActivityNodeActivation* représentée par la propriété *schedulingList*. Cette liste contient toutes les actions prêtes à être exécutées c'est-a-dire des actions qui possèdent tous leurs jetons de données et de contrôle.

L'ordonnanceur offre plusieurs opérations pour contrôler les exécutions des actions. Ces opérations sont appelées dans le corps de l'opération *start()* qui démarre le comportement de l'ordonnanceur dans le moteur d'exécution de fUML. L'opération *selectNextAction()* détermine la prochaine action à exécuter en extrayant un élément de *schedulungList* en fonction d'une politique d'ordonnancement. L'opération *updateSchedulingList()* détermine les successeurs potentiels de la dernière action exécutée puis les ajoute dans liste de l'ordonnanceur.

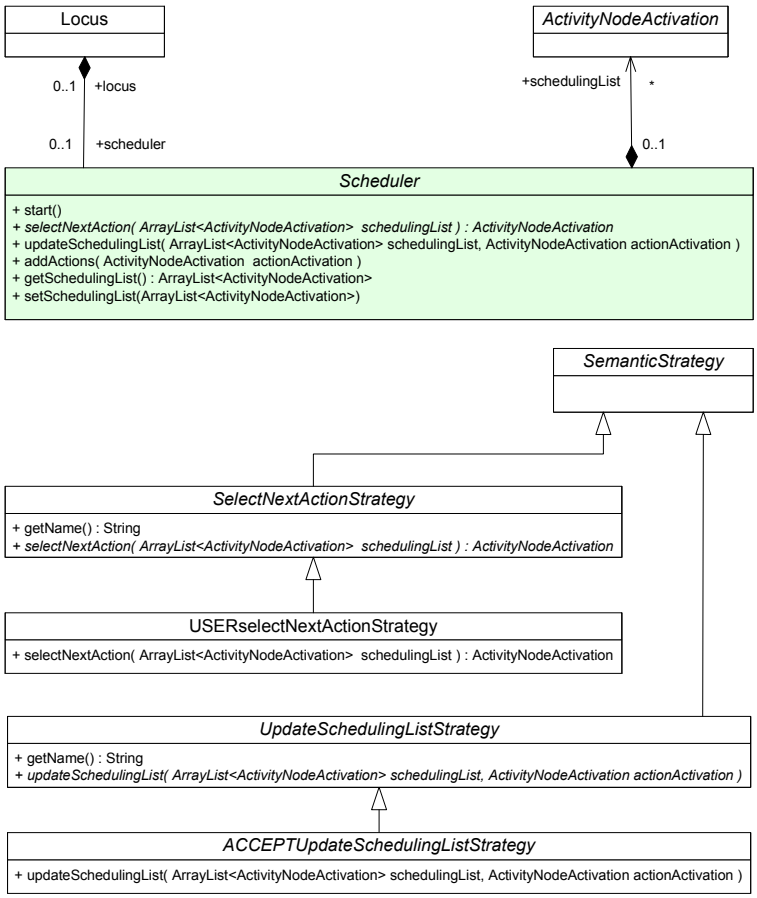


FIGURE 4.1 – Description de l'ordonnanceur dans le modèle d'exécution de fUML

Afin d'insérer le comportement de l'ordonnanceur dans le modèle d'exécution de fUML et de lui donner la main pour déterminer quelle action exécuter, nous avons modifié le comportement responsable de la propagation d'appel d'opération. Ce comportement est défini dans les classes *ActivityNodeActivation* et *ActivtyEdgeInstance* et capturé dans les deux cas par l'opération *sendoffer()*.

Pour illustrer le problème, nous présentons un exemple dans la figure 4.2. Cet exemple explique le mécanisme de propagation d'appels et de jetons dans fUML sur deux actions. La partie supérieure de la figure contient le modèle syntaxique des deux actions reliées par un arc de flot de données (couleur verte) et un arc de flot de contrôle (couleur bleu). La partie inférieure de la figure représente les classes du modèle d'exécution qui correspondent à ce

modèle syntaxique. La classe *ActivityNodeActivation* encapsule la sémantique d'exécution de chaque action. La classe *ActivtyEdgeInstance* représente dans le modèle d'exécution des arcs connectant les actions du modèle syntaxique. Elle *encapsule* une partie du comportement de la propagation d'appel ainsi que le comportement responsable de la propagation des jetons. L'ensemble des opérations présentées dans la figure est restreint afin de simplifier l'exemple.

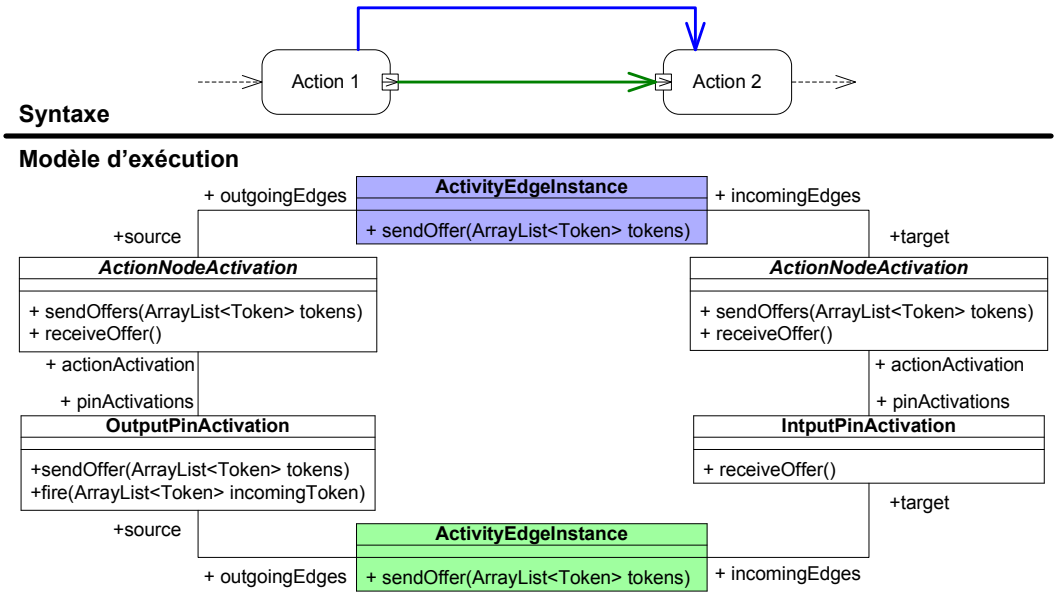


FIGURE 4.2 – Mécanisme de propagation d'appel d'opérations et de jetons dans fUML

Le diagramme de séquence des appels d'opérations nécessaires pour l'exécution des deux actions de l'activité est présenté dans la figure 4.3. Les étapes de l'exécution sont donc :

1. L'opération *receiveOffer()* est appelé sur la première action pour déclencher l'exécution.
2. à la fin de l'exécution de chaque nœud, un appel vers l'opération *sendOffer()* des instances d'arc en sortie du nœud est effectué sauf dans le cas de la classe *InputPinActivation*.

Nous constatons que les deux classes *ActivityEdgeInstance* jouent un rôle important dans la propagation de l'exécution. Plus précisément, le body de leur opération *sendOffer()* est devisé en deux parties :

- ➢ La première partie copie les jetons dans les nœuds successeurs.
- ➢ La deuxième partie, appel l'opération *receiveOffer()* sur le nœud cible pour enchainer l'exécution.

Notre but est de donner la main à l'ordonnanceur après l'exécution de chaque action. Pour cela, il est nécessaire de rompre l'enchainement des appels d'opérations au niveau de l'opération *sendOffer()* de la classe *ActivtyEdgeInstance*. Les jetons de données et de contrôle sont toujours fournis aux actions successeurs, mais l'exécution de ces nœuds n'est pas déclenchée. Concrètement, la modification est réalisée par la suppression de l'appel de

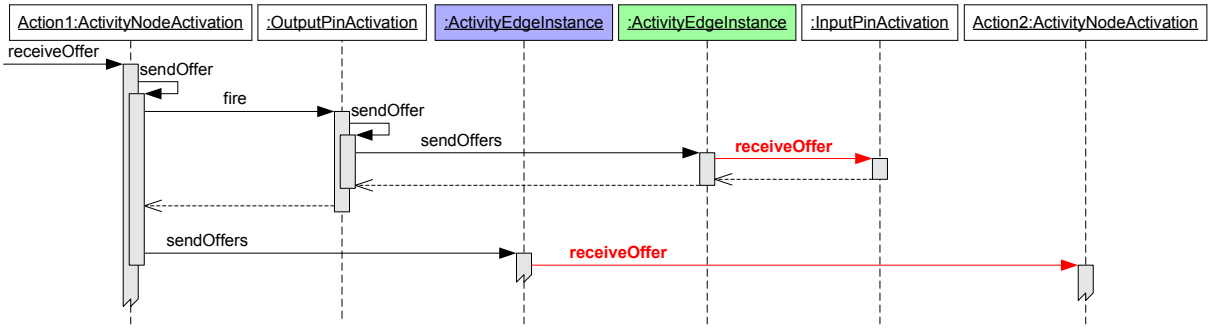


FIGURE 4.3 – Diagramme de séquence de l'exécution des deux actions

l'opération *receiveOffer()* (colorée en rouge sur le diagramme de séquence) dans le corps de l'opération *sendOffer()* de la classe *ActivtyEdgeInstance*. De cette manière, une action n'enchaine pas l'exécution sur l'action suivante mais redonne la main à l'ordonnanceur.

Afin de capturer différentes politiques d'ordonnancement, nous avons exploité le patron de conception de stratégie utilisé dans le modèle d'exécution. Pour cela, nous avons introduit la classe abstraite *SelecNextActionStrategy*. Cette classe fournie l'opération *selectNextAction()* dont le comportement sera un algorithme d'ordonnancement. En effet, définir une nouvelle politique revient à raffiner la classe *SelectNextActionStrategy* et définir une classe concrète dans laquelle l'opération *selectNextAction()* sera surchargée par le comportement d'une politique d'ordonnancement particulière . Par exemple, dans la figure 4.1, la classe concrète *USERSelectNextActionStrategy* implémente une politique utilisateur. Cette politique donne la main à l'utilisateur pour choisir lui-même l'action à exécuter.

La figure 4.4 présente un diagramme de séquence qui explique l'interaction de l'ordonnanceur avec une action. Les différentes étapes réalisées par l'ordonnanceur sont résumées par :

- Le comportement de l'ordonnanceur est lancé juste après l'initialisation du moteur d'exécution par un appel à l'opération *start()*.

- Une fois l'ordonnanceur déclenché, l'opération *selectNextAction()* est exécutée. Cette opération choisit une action depuis la liste des actions en fonction d'une politique particulière. L'implémentation de l'opération *selectNextAction()* consiste à déléguer le choix de l'action à une classe stratégie concrète encapsulant l'algorithme d'ordonnancement.

- Après la sélection d'une action, l'opération *doAction()* est appelée pour exécuter le comportement propre à l'action. Par la suite, l'opération *sendOffer()* propage les jetons aux prochaines actions successeurs puis rend la main à l'ordonnanceur.

- L'opération *updateSchedulingList()* est appelée pour mettre à jour la liste de actions en ajoutant les différentes actions successeurs. La prochaine action à exécuter est sélectionnée par un autre appel de l'opération *selectNextAction()*.

En effet, ces étapes seront répétées jusqu'à ce qu'il ne reste aucune action dans la liste d'ordonnancement.

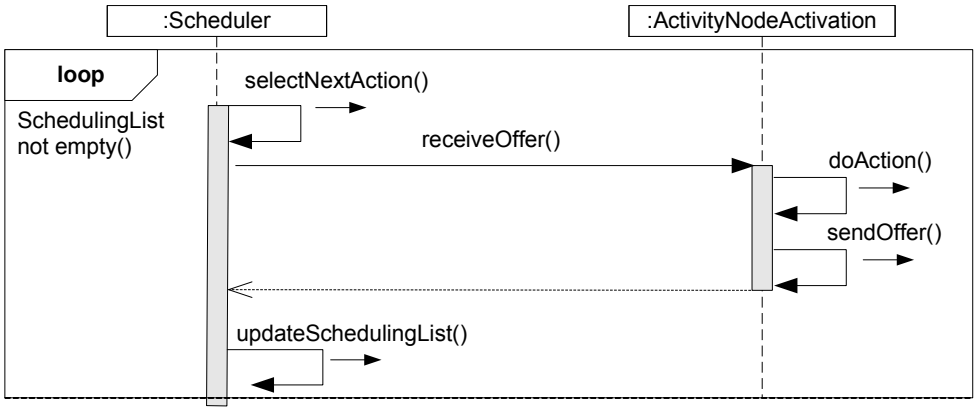


FIGURE 4.4 – L'interaction de l'ordonnanceur avec une action

Il est à noter que la détermination des actions successeurs et la mise à jour de la liste d'actions pourraient avoir des comportements différents en fonction stratégies particulières. Afin de pouvoir spécifier ces comportements, nous avons utilisé également le patron de stratégie du modèle d'exécution. Pour cela, nous avons introduit la classe abstraite *UpdateSchedulingListStrategy*. Cette classe permet d'implémenter chaque mécanisme de mise à jour en fournissant un raffinement concret de cette classe et en surchargeant l'opération *UpdateSchedulingList()* (voir la figure 4.1).

Pour démontrer l'apport de l'ordonnanceur dans le modèle d'exécution de fUML, nous allons présenter les résultats de simulation de deux objets concurrents avec et sans ordonnanceur. L'exemple suivant décrit deux classes actives qui communiquent par envoi de signal asynchrone. La figure 4.5, illustre le modèle structurel et le modèle du comportement correspondant qui vont être simulés. Les instances des deux classes actives *C1* et *C2* vont échanger les signaux *Signal1* et *Signal2*. Les comportements de *C1* et *C2* sont décrits respectivement par les activités *C1Behavior* et *C2Behavior*. L'instance de la classe *C1* envoie le signal *Signal1* à l'instance de *C2*, puis attend la réception du signal *Signal2* qui va être envoyé par l'instance de *C2*. Du coté de l'instance de *C2*, elle attend la réception du signal *Signal1* envoyé par l'instance de *C1*. À sa réception, elle envoie le signal Signal2 à l'instance de *C1*.

Comme tout programme objet, amorcer une exécution nécessite une fonction *Main*. Dans le contexte de fUML, la simulation d'un modèle est toujours lancée à partie d'une activité *Main*. Cette activité est l'équivalent de la fonction *Main* d'un programme. Elle décrit la manière dont les objets d'une application seront instanciés et démarrés. Pour notre exemple, cette fonction *Main* est modélisée dans l'activité présentée par la figure 4.6. Les actions encadrées en rouge sont responsables de l'instanciation des objets c1 et c2. La partie encadrée en vert démarre les comportements des objets actifs capturés par les activités de la figure 4.5. Nous avons supposé que le comportement de *C2* démarre avant le comportement *C1*. Cela est matérialisé par un arc de flot de contrôle sortant de l'action *StartObjectBehaviorAction c2*

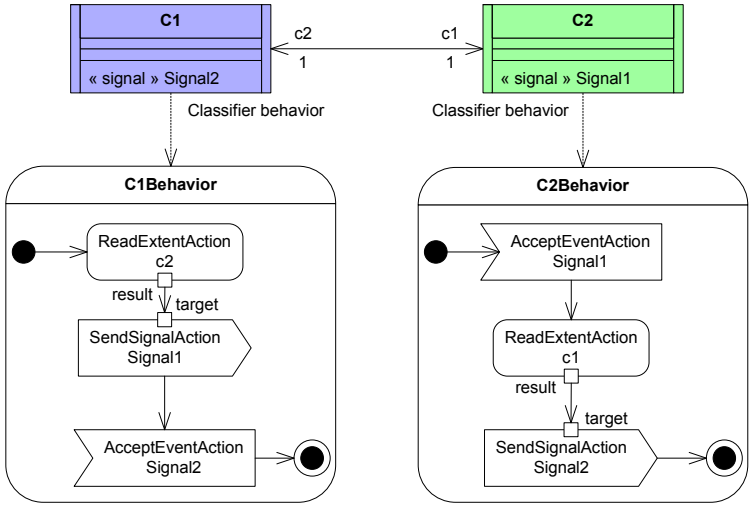


FIGURE 4.5 – Modèle fUML d'un système asynchrone simple

vers l'action *StartObjectBehaviorAction c1*.

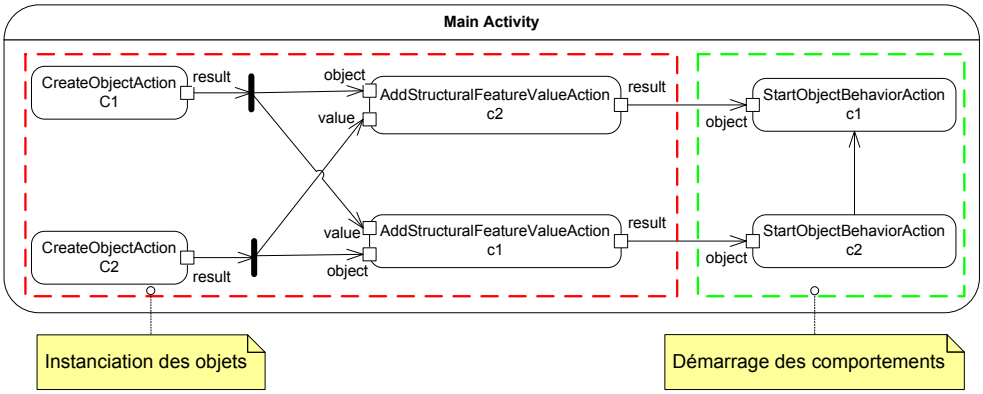


FIGURE 4.6 – L'activité Main du système asynchrone

Dans l'exemple, les classes C1 et C2 sont dites « actives ». , On parle d'objects actifs pour les instances de classes actives. Dans fUML, l'objet actif est un élément fondamental pour la modélisation de la concurrence. Il se caractérise par un comportement autonome qui interagit avec les autres objets de son environnement. Ce comportement est spécifié dans le *classifierBehavior* de la classe active. Il décrit comment l'objet actif envoi et traite les signaux reçus des autres objets. Une fois un objet actif est instancié, son comportement est démarré en utilisant l'action *StartObjectBehaviorAction*.

Afin de faciliter la compréhension de la trace d'exécution de l'exemple, nous présentons succinctement le modèle sémantique de l'objet actif de fUML. La figure 4.7 présente l'ensemble des classes qui spécifie la sémantique de l'objet actif dans le modèle d'exécution. Il est caractérisé principalement par deux classes :

- La classe *ClassifierBehaviorExecution* gère l'exécution du comportement définie dans le *classifierBehavior* de la classe active.

➢ La classe *ObjectActivation* fournie les primitives nécessaires pour gérer les communications notamment asynchrones. L'opération *send()* spécifie la réception d'un signal. Les signaux reçus sont stockés dans la liste des événements *eventPool*. La notification d'une réception est capturée par l'opération _ *send()*. Les autres opérations sont utilisées essentiellement pour récupérer un signal de la liste des événements et lancer le traitement des réceptions.

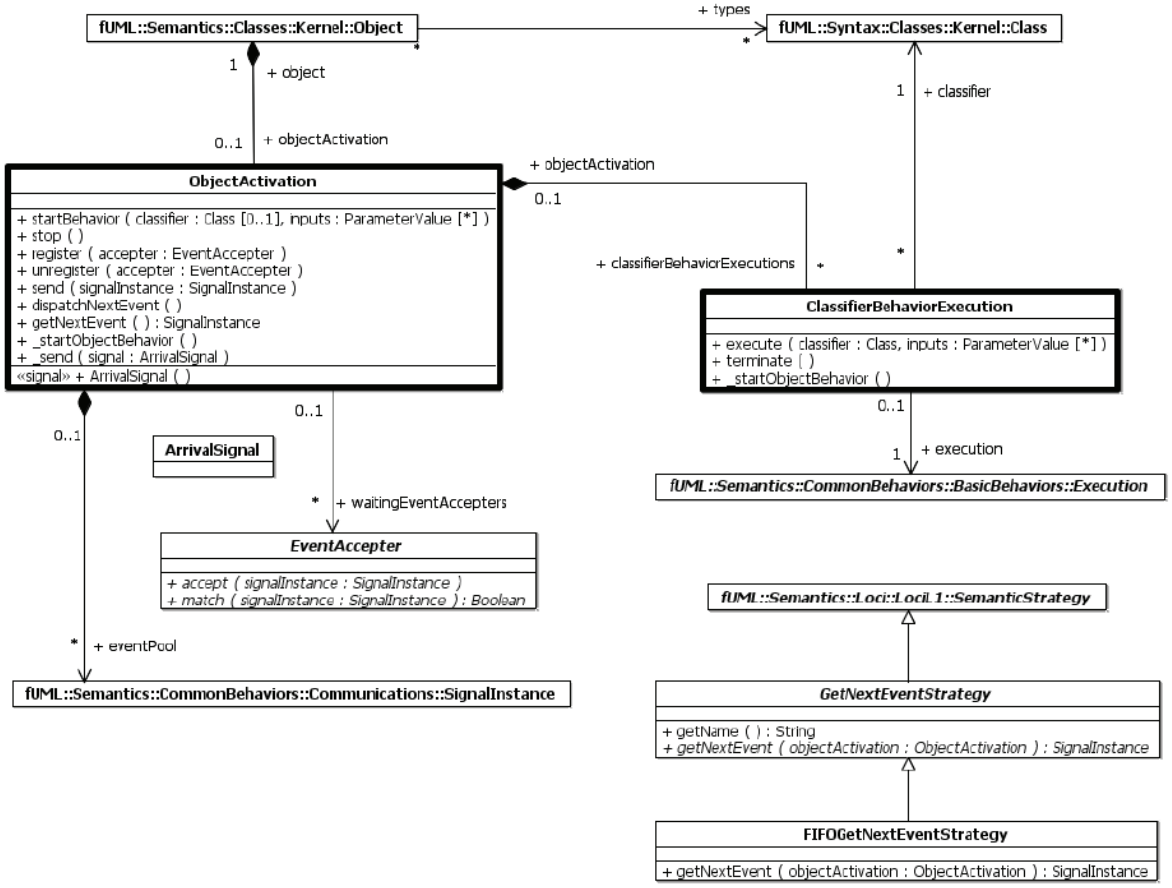


FIGURE 4.7 – Modèle sémantique de l'objet actif

La figure 4.8 illustre la trace d'exécution en respectant le modèle d'exécution de fUML avant l'introduction de l'ordonnanceur. Afin de simplifier le diagramme, la séquence d'exécution de l'activité *Main* n'est pas présentée. On rappelle que toute l'exécution est réalisée séquentiellement sur un seul thread. Les lignes de vie apparaissant dans le diagramme de séquence représentent les instances des classes du modèle d'exécution. Les interactions entre ces lignes de vie montrent comment le modèle de la figure 4.5 est simulé. Les instances colorées en bleu appartiennent à l'objet *c1* tandis que les instances colorées en vert représentent l'objet *c2*. Les instances de la classe *ClassifierBehaviorExecution* représentent respectivement l'exécution des *classifierBehavior* de l'objet *c1* et *c2*.

Comme l'objet *c2* démarre avant *c1*, le classifierBhevaior de *c2* est lancé en premier. Les différentes étapes d'exécution respectent l'ordre suivantes :

≻ Lancement de l'activité *C2Behavior*. L'exécution commence par l'action *AcceptEventAction*. Cette action capture l'attente de *c2* pour la réception d'une occurrence du signal *Signal1*. Dans cette étape, au lieu du blocage de l'objet *c2* pour attendre le signal, l'exécution revient à l'activité *Main*.

≻ Lancement de l'activité *C1Behavior*. Cette activité exécute l'action *SendSignalAction* qui ajoute une instance du signal *Signal1* dans la liste des événements associée à la classe *ObjectActivation* de l'objet *c2*.

≻ Notification de l'objet *c2* de la disponibilité d'un signal. Cette étape retire le signal *Signal1* de la liste des événements puis vérifier la présence d'une attente pour ce signal. Dans cette étape de l'exécution, l'activité *C2Behavior* de *c2* reprend l'exécution.

≻ L'activité *C2Behavior* exécute l'action *SendSignalAction* pour envoyer le signal *Signal2* à l'objet *c1*.

≻ Notification de l'objet *c1* de la disponibilité d'un signal. La non disponibilité d'une attente du signal *Signal2* implique l'arrêt de l'exécution.

Dans ce cas, le signal *Signal2* est perdu et l'exécution de l'activité *C1Behvaior* n'arrive pas jusqu'au bout. Cela entraine l'arrêt de simulation sans terminer l'exécution des deux activités.

Nous présentons maintenant la trace d'exécution de notre exemple en respectant le modèle d'exécution de fUML après l'introduction de l'ordonnanceur. La politique d'ordonnancement adoptée est une politique *USER*. L'algorithme d'ordonnancement dans ce cas fournit une interface à l'utilisateur. L'utilisateur peut choisir donc l'action à exécuter parmi les actions présentes dans la liste des actions, après chaque exécution d'action. Cette politique d'ordonnancement est implémentée dans la classe USERSelectNextActionStrategy, présentée dans la figure 4.1.

La figure 4.9 illustre le diagramme de séquence de l'exécution avec ordonnancement. L'ordonnanceur est déclenché par l'opération *start*. Une fois démarré, il donne la main à l'utilisateur pour choisir une action à exécuter. L'ordre des actions que nous avons sélectionnées est choisi dans le but de terminer l'exécution des deux activités de l'exemple. La figure 4.10 est complémentaire au diagramme de séquence, elle présente l'état de la liste des actions et l'action que nous avons choisie après chaque exécution d'une action. Afin de simplifier la trace d'exécution, on donne l'état de la liste après l'exécution de l'activité *Main*, on commence à partir de l'initialisation des activités *C1Behavior* et *C2Behavior*. On constate que dans ce cas, nous avons pu éviter la perte des signaux. De plus, les deux activités ont été complètement exécutées.

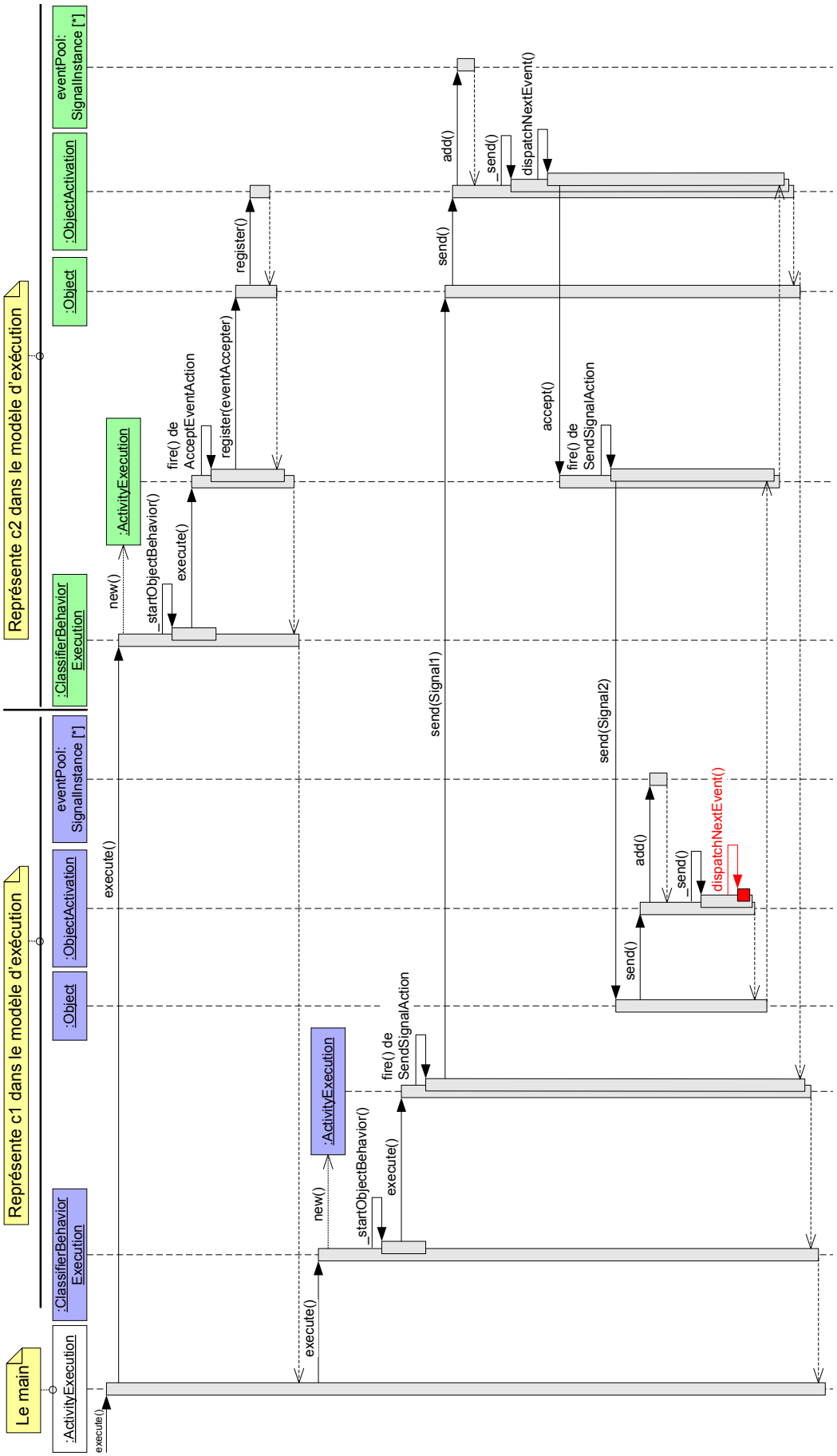


FIGURE 4.8 – Trace d'exécution sans ordonnanceur

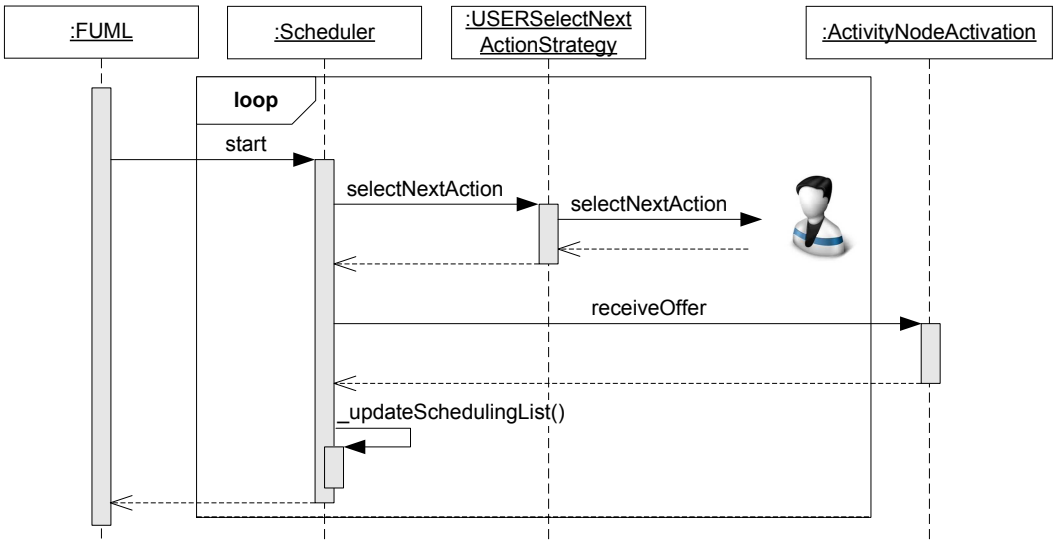


FIGURE 4.9 – Trace d'exécution avec ordonnanceur

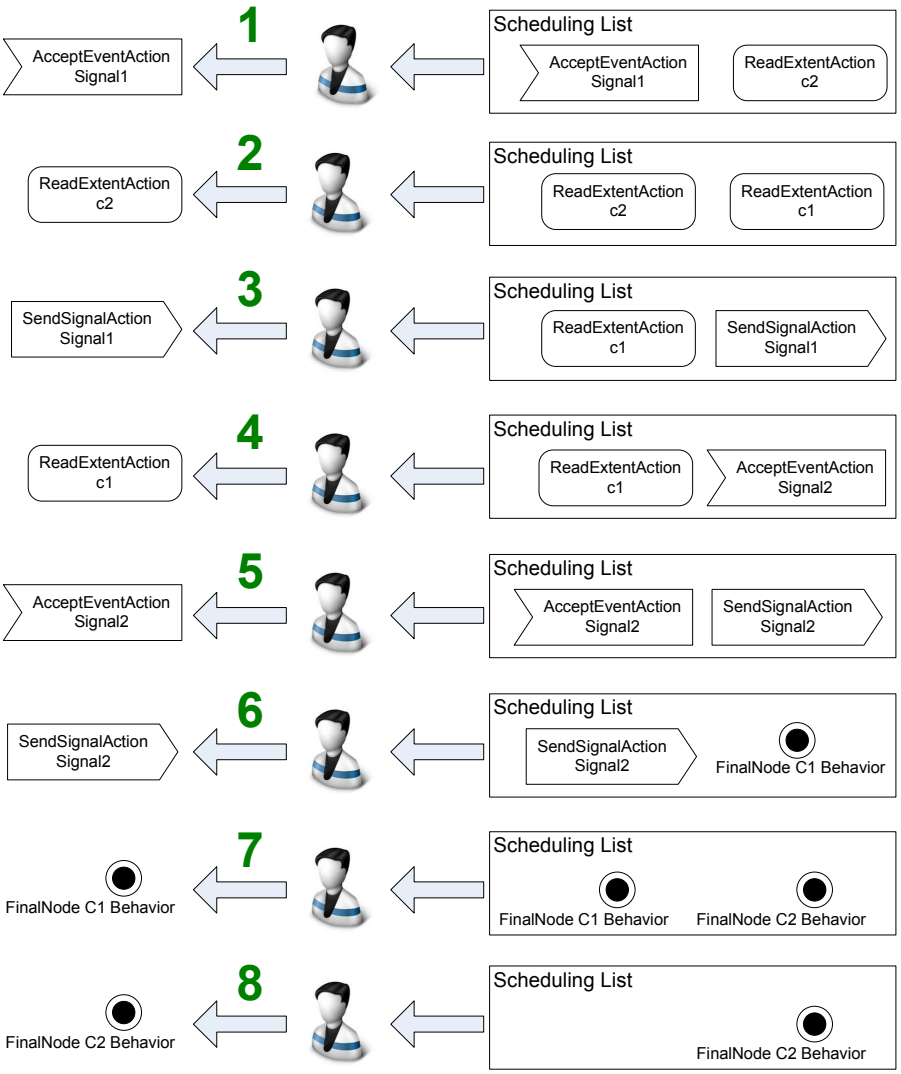


FIGURE 4.10 – L'ordre des actions sélectionnées par l'utilisateur

1.1 Conclusion

L'objectif de cette section était de résoudre le problème d'ordonnancement dans fUML. Pour cela, nous avons introduit un ordonnanceur générique dans le modèle d'exécution de fUML. Cet ordonnanceur offre les mécanismes permettant de contrôler les exécutions des différentes actions et de définir plusieurs politiques d'ordonnancement.

Après avoir défini et validé expérimentalement les mécanismes d'ordonnancement dans fUML, il faut désormais s'assurer que ces mécanismes peuvent être utilisés pour interpréter les caractéristiques temporelles des modèles de STRE. La section suivante vise à étendre le modèle d'exécution de fUML afin de supporter la notion du temps.

2 Le temps dans fUML

Comme nous l'avons évoqué dans l'état de l'art, fUML ne fixe aucune hypothèse sur la manière dont les informations sur le temps sont capturées ainsi que sur les mécanismes qui les traitent dans le modèle d'exécution. Cela laisse la possibilité d'adopter différents modèles de temps (discret ou continu). Dans cette étude, nous nous sommes focalisés sur un modèle de temps discret. L'objectif est de produire des traces d'exécutions représentatives des caractéristiques temps réel d'un modèle applicatif, notamment des périodes, des échéances, des temps d'exécution, etc. La partie supérieure de la figure 4.11 montre un diagramme d'activité dans lequel chaque action est annotée par son temps d'exécution. La partie inférieure de la figure 4.11 illustre les traces d'exécutions potentielles dans le cas d'une exécution séquentielle et parallèle.

Dans le cas séquentiel, le temps d'exécution globale est la somme des temps d'exécution de chaque action. L'action 1 démarre à t=0 sec et finit à t=2 sec, l'action 2 démarre à t=2 sec et finit à t=6 sec. L'action 3 démarre à t=6 sec et finit à t=14 sec. Dans le cas parallèle, l'action 1 et l'action 2 démarrent en parallèle à t=0 sec, l'action 3 démarre à l'instant où l'action ayant le plus grand temps d'exécution termine son exécution, elle démarre donc après l'action 2 à t=4 sec. L'exécution globale de l'activité termine à t=12 sec. Ces traces indiquent l'instant de déclenchement et de fin d'exécution de chaque action. C'est ce type d'information que nous voulons observer dans les traces d'exécutions. Dans le contexte d'une application temps réel, nous pouvons ainsi détecter si une échéance est respectée ou non par exemple.

Pour ajouter la notion du temps, nous avons étendu fUML au niveau syntaxique ainsi qu'au niveau du modèle d'exécution. Au niveau syntaxique, les stéréotypes sont utilisés pour capturer les caractéristiques temporelles d'un modèle, par exemple le stéréotype RtSpecification du profil MARTE [34] pourrait être utilisé. Cette extension est détaillée dans la section suivante. Au niveau sémantique, des mécanismes sont ajoutés au modèle d'exécution afin de prendre en compte explicitement la sémantique du temps. En effet, nous avons introduit une entité permettant d'une part l'étiquetage temporel des exécutions, c'est-a-dire la construction des pas de temps des exécutions. D'autre part, elle permet de déclencher des exécutions contraintes par le temps, par exemple pour le cas de tâches périodiques. Cette entité est une horloge représentée dans le modèle d'exécution par la classe *Clock*. La figure 4.12, illustre la description de cette classe dans le modèle d'exécution de fUML.

Les services offerts par l'horloge pour manipuler et construire le temps sont :

- La propriété *timeValue* modélise les valeurs du temps dans le modèle d'exécution. C'est un compteur utilisé pour construire le temps des exécutions.
- L'opération *getTime()* est utilisée pour récupérer les valeurs courantes du temps.
- L'opération *setTime()* est utilisé pour initialiser l'horloge par une valeur initiale de temps

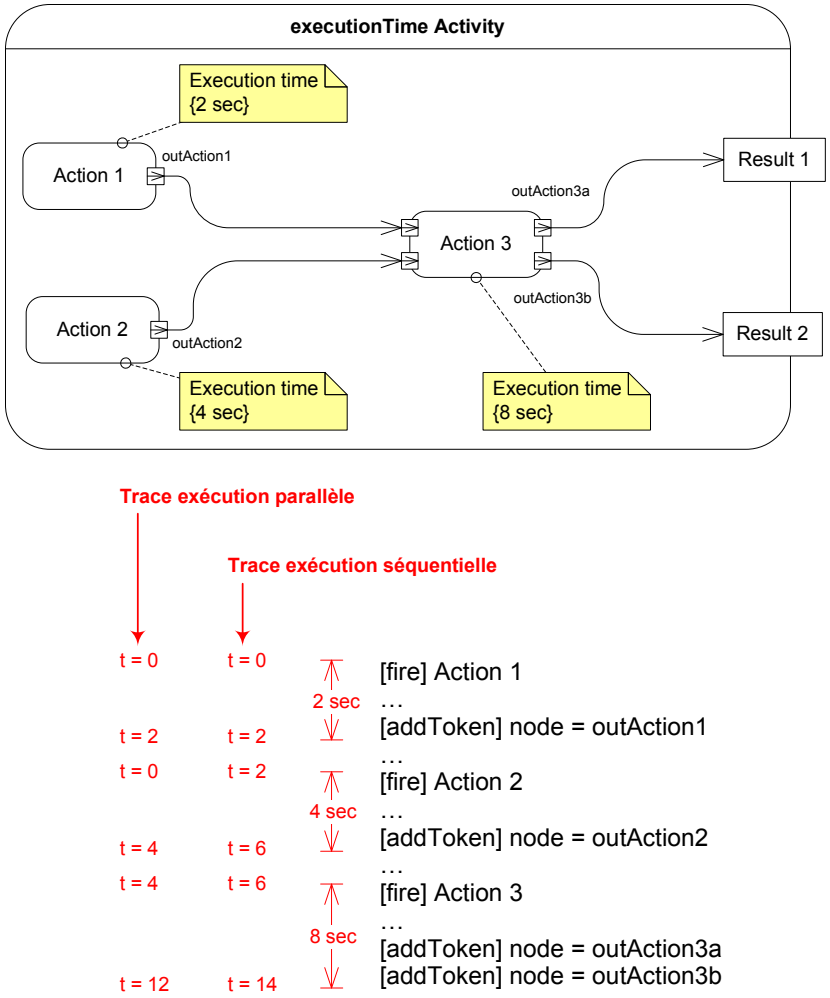


FIGURE 4.11 – Trace d'exécution contenant des informations temporelles

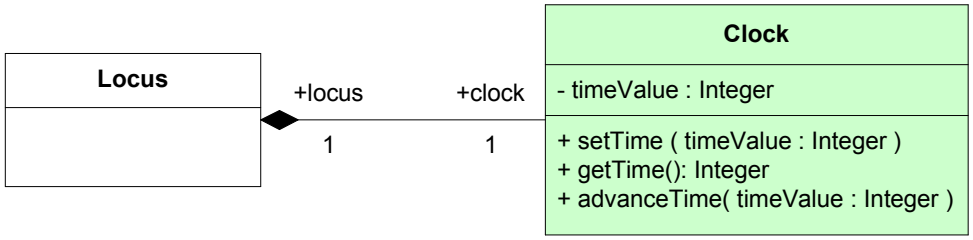


FIGURE 4.12 – L'horloge du modèle d'exécution de fUML

➢ L'opération *advanceTime()* est utilisé pour avancer le temps dans le modèle d'exécution par un pas de temps, par exemple le temps d'exécution d'une action.

Concrètement, générer une trace d'exécution contenant des informations temporelles revient à combiner les différents services de l'horloge soit dans l'algorithme global de l'ordonnanceur soit dans l'algorithme de la politique de l'ordonnancement (l'opération *selectNextAction*). Dans le cas de l'étiquetage temporel des exécutions, seule la modification de l'algorithme de l'ordonnanceur global est nécessaire. Les étapes à rajouter sont :

1. Au début de l'exécution, la valeur initiale du temps est initialisée par un appel à l'opération *setTime()*.
2. Après chaque exécution d'une action, l'opération *advanceTime()* est appelée pour incrémenter la valeur du temps par le temps d'exécution de l'action exécutée.

Dans le cas du déclenchement des exécutions contraintes par le temps, la modification de l'algorithme globale de l'ordonnancement et de l'algorithme de la politique d'ordonnancement est nécessaire. Les étapes à rajouter dans le comportement de l'ordonnanceur sont les mêmes que les étapes rajoutées précédemment. Quant à la politique d'ordonnancement, son algorithme doit utiliser la valeur courante du temps pour choisir la prochaine action à exécuter. Cela est réalisé dans le corps de l'opération *selectNextAction*.

La figure 4.13, montre les modifications effectuées pour l'étiquetage temporel (a) ainsi que pour déclencher des actions périodiquement (b). Les carrés verts représentent les étapes additionnelles.

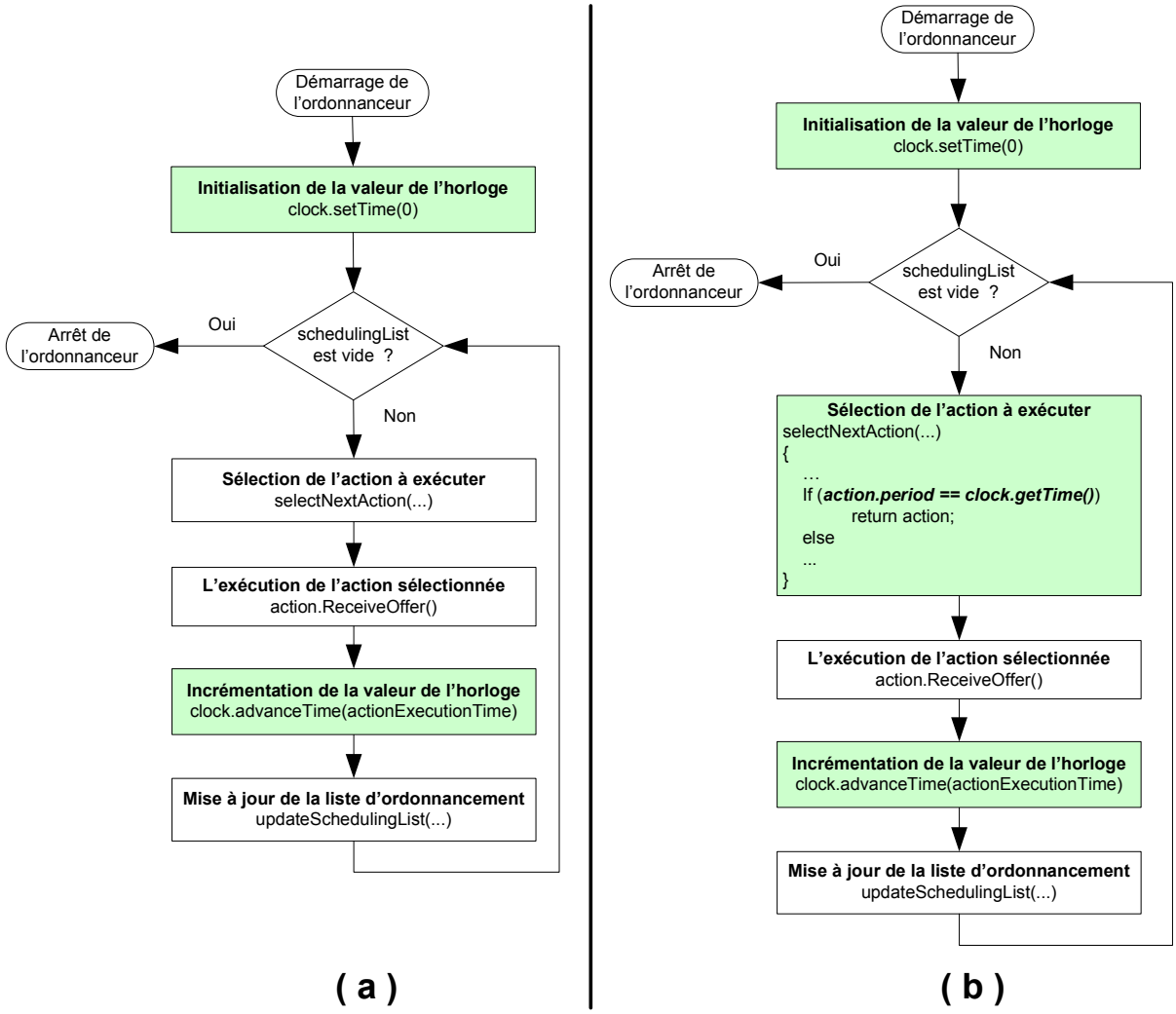


FIGURE 4.13 – Comportement modifié de l'ordonnanceur pour la génération de traces d'exécutions temporelles

2.1 Conclusion

Dans cette section, l'objectif était d'ajouter la notion du temps à fUML. Nous avons introduit une horloge dans le modèle d'exécution de fUML pour capturer la sémantique du temps. Cette horloge se base sur un modèle de temps discret. Elle offre les mécanismes qui permettent de manipuler et de construire le temps des exécutions. Nous avons montré comment modifier l'ordonnanceur afin de supporter les exécutions temporisées.

A ce stade de l'étude, l'horloge seule ne permet pas de générer des traces d'exécution temporisées, il est nécessaire d'ajouter les mécanismes permettant de (a) capturer les informations temporelles au niveau syntaxique et (b) d'exploiter explicitement ces informations au niveau du modèle d'exécution. Le but de la section suivante est d'étendre fUML afin d'utiliser les profils pour répondre aux points (a) et (b).

3 Les profils dans fUML

La prise en compte des profils par fUML nécessite l'extension du sous-ensemble syntaxique et du modèle d'exécution. D'une part, le sous-ensemble syntaxique doit fournir les éléments nécessaires pour appliquer un profil et annoter un modèle par des stéréotypes. D'autre part, le modèle d'exécution doit inclure les éléments permettant de capturer les extensions sémantiques apportées par l'application des stéréotypes.

3.1 L'extension syntaxique

Dans la norme de fUML [86], le modèle d'exécution spécifie une sémantique pour un sous ensemble syntaxique d'UML. Ce sous-ensemble n'inclut aucun élément syntaxique qui permet d'appliquer un profil. Le mécanisme d'application de stéréotype reste par ailleurs un problème ouvert dans le standard d'UML2 [92] dont la solution est fixée au niveau des implémentations des outils de modélisation.

Pour étendre syntaxiquement fUML d'une manière efficace, nous avons utilisé une implémentation du méta-modèle d'UML basée sur EMF [67]. Ce choix est motivé par :

- ≻ Cette implémentation fournie une solution pour le mécanisme d'application des stéréotypes.
- ≻ Cette implémentation facilite toutes extensions syntaxiques de fUML. Dans le cas normal, une extension syntaxique revient à ajouter un nouvel élément dans fUML puis ajouter l'élément qui définit sa sémantique dans le modèle d'exécution. Dans le cas de l'implémentation basée sur EMF du méta-modèle d'UML, le sous ensemble syntaxique utilisé est délimité par les éléments du modèle d'exécution. Ajouter un nouvel élément syntaxique, revient simplement à ajouter l'élément qui lui correspond dans le modèle d'exécution puis définir sa sémantique.
- ≻ Dans cette étude on veut développer un plugin de simulation de modèle pour le domaine du temps réel intégré à l'outil de modélisation Papyrus. Comme cet outil est basé sur une implémentation EMF du méta-modèle d'UML. Cela évite un effort supplémentaire de développement d'une nouvelle implémentation du méta-modèle de fUML et de l'adaptation à celle utilisée par Papyrus.

La figure 4.14 explique le principe d'application de profil et des stéréotypes dans Papyrus. La partie (a) de la figure montre un profil simple. Il définit le stéréotype *Device* qui sera appliqué sur des instances de la méta-classe *Class*. La partie (b) de la figure illustre le modèle d'instance du profil créé par Papyrus afin d'être appliqué sur un modèle. La partie (c) de la figure montre l'application du stéréotype *Device* sur une classe. La partie (d) de la figure illustre le modèle d'instance du modèle et de l'application du stéréotype. Les étapes de création du modèle d'instance sont les suivantes :

1. D'abord les instances qui correspondent au modèle sont créés (une instance de Package

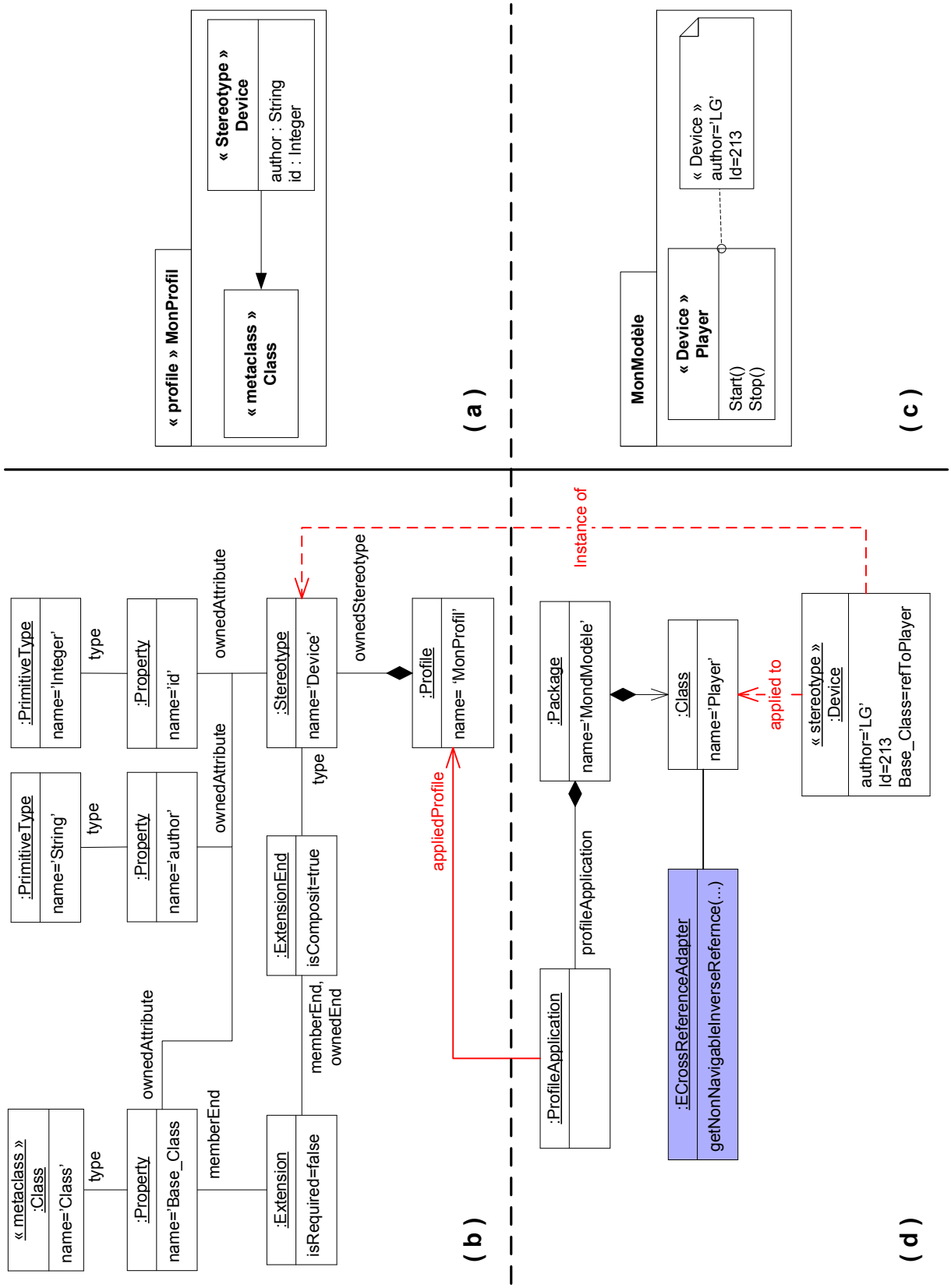


FIGURE 4.14 – Mécanisme d'application des stéréotype dans Papyrus

et une instance de Class).

2. L'application du profil *MonProfil* sur le package *MonModèle* est concrétisée par la création d'une instance de ProfileApplication. Cette classe effectue le lien avec le modèle et le profil appliqué, elle donne accès aux stéréotypes applicables sur les éléments du modèle.
3. L'application du stéréotype *Device* est représentée par une instance du stéréotype *Device* du modèle d'instance du profil.

Conformément à la norme d'UML2 [18], seule l'instance qui représente l'application du stéréotype connait l'élément syntaxique auquel elle est attachée. En conséquence, l'élément syntaxique ne peut pas connaitre le stéréotype appliqué sur lui. Ceci pose particulièrement un problème dans fUML si on veut récupérer les stéréotypes appliqués sur des éléments syntaxiques puis exploiter leurs informations dans le modèle d'exécution. Afin de permettre aux éléments syntaxiques de récupérer leurs applications des stéréotypes, Papyrus/EMF attache une classe supplémentaire (*ECrossReferenceAdapter* de la partie (d) de la figure 4.14) à chaque élément syntaxique. Cette classe n'appartient pas au méta-modèle d'UML. Elle offre les services nécessaires permettant aux éléments syntaxiques de récupérer leurs applications de stéréotypes.

3.2 L'extension sémantique

L'extension sémantique consiste à ajouter dans le modèle d'exécution les éléments nécessaires pour capturer les extensions sémantiques impliquées par l'utilisation d'un profil. Ces éléments doivent exploiter les informations des stéréotypes pour le paramétrage du modèle d'exécution afin de simuler les modèles selon une sémantique particulière. La figure 4.15 illustre l'ensemble des classes que nous avons introduit dans le modèle d'exécution.

➢ **La classe *StereotypeValue* :** Cette classe représente l'application de stéréotype dans le modèle d'exécution. C'est une classe de type visiteur. Elle encapsule les valeurs marquées du stéréotype dans une table de hachage (*taggedValues*) afin de faciliter l'accès à ces valeurs. En effet, pour chaque stéréotype appliqué au niveau syntaxique, une instance de *StereotypeValue* est créée dans le modèle d'exécution puis associée à l'instance de la classe *AcitivtyNodeAcitivation* qui correspond au nœud syntaxique sur lequel le stéréotype est appliqué.

➢ **La classe *StereotypeFactory* :** Cette classe fait partie des éléments qui constituent le moteur d'exécution. Son rôle principal est la création des instances de la classe *StereotypeValue*, il est assuré par l'opération *createStereotypeValue*. Le deuxième service fourni par cette classe est la recherche d'une instance de *StereotypeValue* dans une collection d'instances en fonction du nom du stéréotype. Il est réalisé par l'opération *findStereotypeValue*.

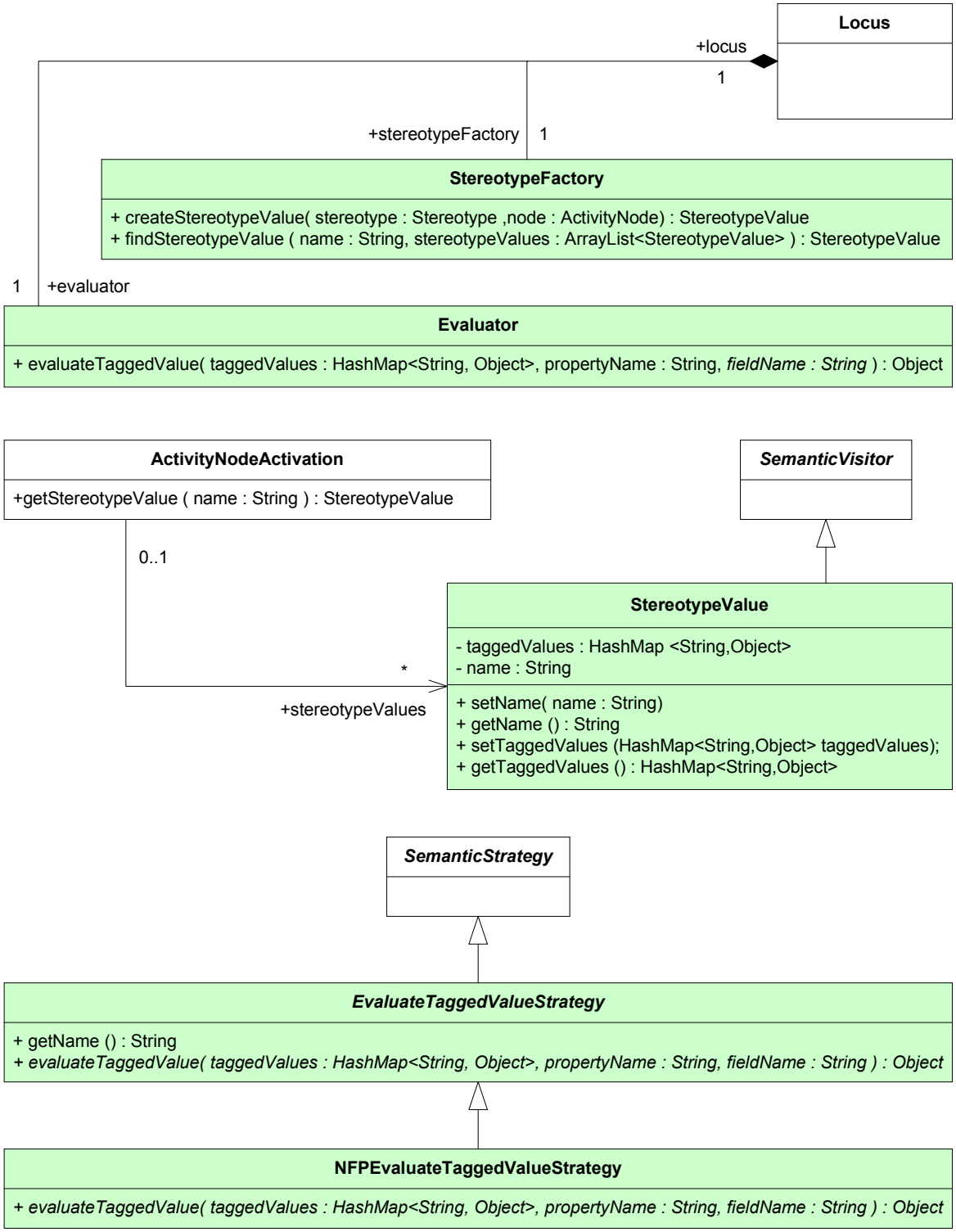


FIGURE 4.15 – Les classes introduites dans le modèle d'exécution pour la prise en compte des profils

➢ **La classe *Evaluator* :** Cette classe est utilisée pour évaluer les valeurs marquées stockées dans la classe *StereotypeValue*. Cette évaluation peut être différente d'une valeur marquée à un autre, en fonction du type des propriétés du stéréotype.

Par exemple l'évaluation d'une valeur marquée de type *NFPDuration* [34] n'est pas effectuée de la même façon qu'une évaluation d'un type primitif. Afin de permettre la spécification de différents mécanismes d'évaluation, nous avons défini l'opération *evaluateTaggedValue* suivant le patron de conception de stratégie. Pour cela, nous introduisons la classe *EvaluteTaggedValueStrategy*. C'est une classe de type de stratégie. Elle sera raffinée pour chaque nouveau mécanisme d'évaluation par la définition une nouvelle classe concrète dans laquelle, l'opération *evaluateTaggedValue* est implémentée. A titre d'exemple, la classe *NFPEvaluteTaggedValue* est une classe concrète qui implémente le mécanisme d'évaluation des type *NFPDuration* du profile MARTE [34].

Les différents services de ces classes peuvent être combinés afin de spécifier une sémantique particulière. Dans la partie suivante, nous présentons un cas d'étude sur lequel nous montrons comment utiliser ces différentes classes pour spécifier la sémantique d'un profil pour le domaine du temps réel.

4 Application et validation : Sémantique d'exécution d'un sous profil de MARTE pour l'analyse d'ordonnancement

Dans les sections précédentes, nous avons présenté en détail différentes extensions à fUML et son modèle d'exécution. La présente section décrit comment utiliser ces extensions pour paramétrer le modèle d'exécution afin de capturer la sémantique d'un profil en particulier. L'exemple choisi est un sous profil de MARTE utilisé notamment dans la méthodologie de modélisation Optimum [93]. Les modèles annotés par ce profil seront simulés pour générer des traces d'exécution représentatives des caractéristiques temps réel, notamment l'occurrence des événements, le temps des exécutions et les périodes. Nous présentons tout d'abord la méthodologie de modélisation Optimum [93], en détaillant les différents stéréotypes utilisés. Nous exposons ensuite un modèle applicatif modélisé en suivant cette méthodologie. Par la suite, nous détaillons la façon dont nous avons paramétré le modèle d'exécution et montrons comment l'approche proposée permet de simuler des modèles annotés par un profil.

4.1 Introduction à la méthodologie de modélisation Optimum

Optimum est une méthodologie de modélisation qui s'appuie sur le profil MARTE. Son objectif est de produire des modèles UML concurrents permettant l'analyse d'ordonnancement. Le modèle final obtenu se base sur des contraintes temps réel durs. Les concepts du profil MARTE utilisés par la méthodologie traitent les notions de modélisation de ressources de plateforme, la modélisation pour l'analyse d'ordonnancement et la modélisation des allocations. Ces stéréotypes sont regroupés dans un profil nommé MARTE4Optimum. Le tableau 4.1 présente l'ensemble des stéréotypes utilisés.

Les stéréotypes de MARTE4Optimum	**Extensions UML**
Alloc : :Allocate	Abstraction
Alloc : :Allocated	CallAction, Property
GRM : :SchedulableResource	Property
GQAM : :GaPlatformResources	Class
GQAM : :GaWorkloadBehavior	Activity
GQAM : :GaWorkloadEvent	AcceptEventAction
SAM : :SaAnalysisContext	Activity
SAM : :SaEndToEndFlow	ActivityPartition
SAM : :SaExecHost	Property
SAM : :SaSharedResource	Property
SAM : :SaStep	CallAction

TABLE 4.1 – Les stéréotypes du profil MARTE4Optimum

Pour commencer la modélisation, la méthodologie exige en entrée des scénarios d'exécutions de type bout-à-bout (end-to-end scenarios) et les exigences temporelles correspondantes. La première étape de modélisation produit un modèle de charge de travail (Workload model). Il est constitué d'un modèle de comportement et d'un modèle de plateforme d'exécution. La deuxième étape de modélisation créée le modèle d'analyse d'ordonnançabilité. Il est obtenu par un mapping des scénarios de type bout-à-bout sur les tâches de la plateforme d'exécution. Dans chaque étape le modèle est raffiné par l'application des stéréotypes du profil MARTE4Optimum. Le modèle final est utilisé principalement pour l'analyse d'ordonnancement. Dans notre cas, nous l'utilisons pour la simulation dans fUML. La figure 4.16 résume les différentes étapes de modélisation dans Optimum.

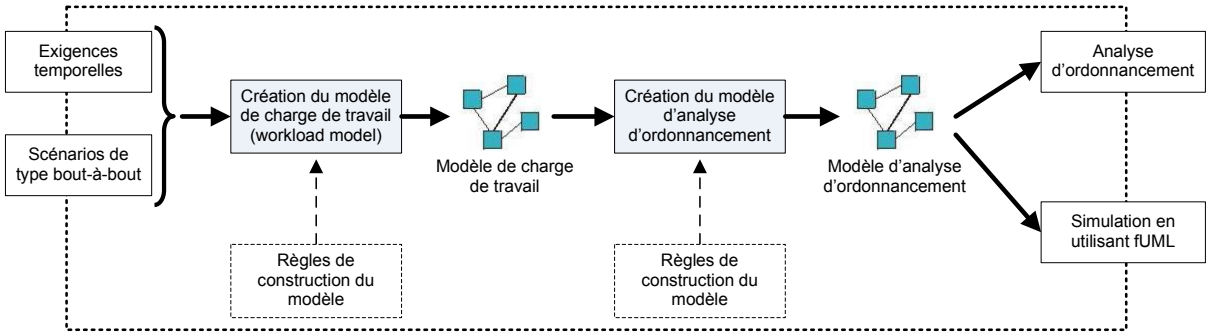


FIGURE 4.16 – Les étapes de modélisation dans Optimum

4.1.1 Le modèle de charge de travail (WorkLoad Model)

Ce modèle est composé d'un modèle de comportement et d'un modèle de plateforme d'exécution. Le modèle de comportement (Workload Behavior) est capturé par les activités d'UML. Il est créé à partir des scénarios sous forme de séquence d'actions déclenchées par des événements externes. Une activité peut avoir plusieurs scénarios. Chaque scénario est défini par les éléments suivants :

- ≻ Les événements sont représentés par des AcceptEventActions.
- ≻ Les actions déclenchées par les événements sont représentées soit par des CallOperationActions ou par des CallBehaviorActions.
- ≻ Un scénario commence toujours par un AcceptEventAction.
- ≻ La dernière action d'un scénario est toujours suivie par un FinalFLowNode.
- ≻ Toutes les actions sont connectées par des arcs de flot de contrôle.
- ≻ Chaque séquence d'action d'un scénario ne doit pas contenir de cycles.
- ≻ Chaque séquence d'action est regroupée dans une ActivityPartition.

Après la définition des différents scénarios, le modèle est enrichi par l'application des stéréotypes suivants :

➢ Le diagramme d'activité est annoté par le stéréotype *GaWorkloadBehavior*.

➢ Chaque *AcceptEventAction* est annotée par le stéréotype *GaWorkloadEvent* dont la politique d'arrivée des événements (périodique, apériodique, sporadique, etc.) est définie dans la propriété *arrivalPattern*.

➢ Chaque *ActivityParition* est annotée par le stéréotype *SaEndToEndFlow*. L'échéance de chaque scénario est spécifiée dans la propriété *end2EndD*.

➢ Chaque action de type *CallOperationAction* ou *CallBehaviorAction* est annotée par le stéréotype *SaStep*, dans lequel le temps d'exécution est spécifié dans la propriété *execTime*.

Quant au modèle de plateforme d'exécution, il définit les unités de calcul (processeurs) et les ressources d'exécution (threads). Dans la première étape de la méthodologie ce modèle n'est qu'une vue abstraite de la plateforme d'exécution. Les éléments de la plateforme d'exécution sont modélisés par des propriétés UML, regroupées dans un classifier UML annoté par le stéréotype *GaResourcesPlatform*. Le type de la propriété désigne le type de la ressource (processeur ou thread). Le modèle obtenu sera raffiné dans les deuxièmes étapes de la méthodologie.

4.1.2 Le modèle d'analyse d'ordonnançabilité

Ce modèle est obtenu par le raffinement du modèle de plateforme et par un mapping des scénarios sur les ressources d'exécution. Les éléments de la plateforme d'exécution sont enrichis comme suit :

➢ L'unit de calcul (processeur) est annotée par le stéréotype *saExecHost*. On doit spécifier dans la propriété *schedPolicy* l'algorithme d'ordonnancement utilisé. Les valeurs possibles sont *FixedPriority* dans le cas d'un algorithme à priorité fixe et *EarliestDeadlineFirst* dans le cas d'un algorithme à échéance la plus proche.

➢ Chaque ressource d'exécution (thread) est annotée par le stéréotype *SchedulableResource*. Dans ce stéréotype, on doit définir les paramètres d'ordonnancement de la ressource d'exécution qui dépendent de l'algorithme d'ordonnancement choisi. Ces paramètres sont définis dans la propriété *schedParams* sous forme d'expression VSL [34]. Dans le cas d'un algorithme à priorité fixe, la priorité est spécifiée sous la forme de *schedParam :SchedParameters[0..*]=[fp(priority-value)]*. Dans le cas d'un algorithme à échéance la plus proche, l'échéance est définie sous la forme de *schedParam :SchedParameters[0..*]=[edf(deadline-value)]*.

Dans la phase de mapping, les scénarios sont affectés aux ressources d'exécution. L'affectation est réalisée soit par scénario, soit par fonction. Dans le premier cas, tout le scénario est assigné à la même ressource calcul. Dans le deuxième cas, les fonctions du même scénario sont affectées à des ressources d'exécution différentes. En effet, le mapping

est réalisé par l'application du stéréotype *allocated* sur chaque élément d'un scénario. On doit spécifier la ressource d'exécution affectée à chaque fonction dans la propriété *allocatedTo*.

4.2 Cas d'étude

Le modèle présenté dans la figure 4.17 est obtenue par l'application de la méthodologie Optimum.

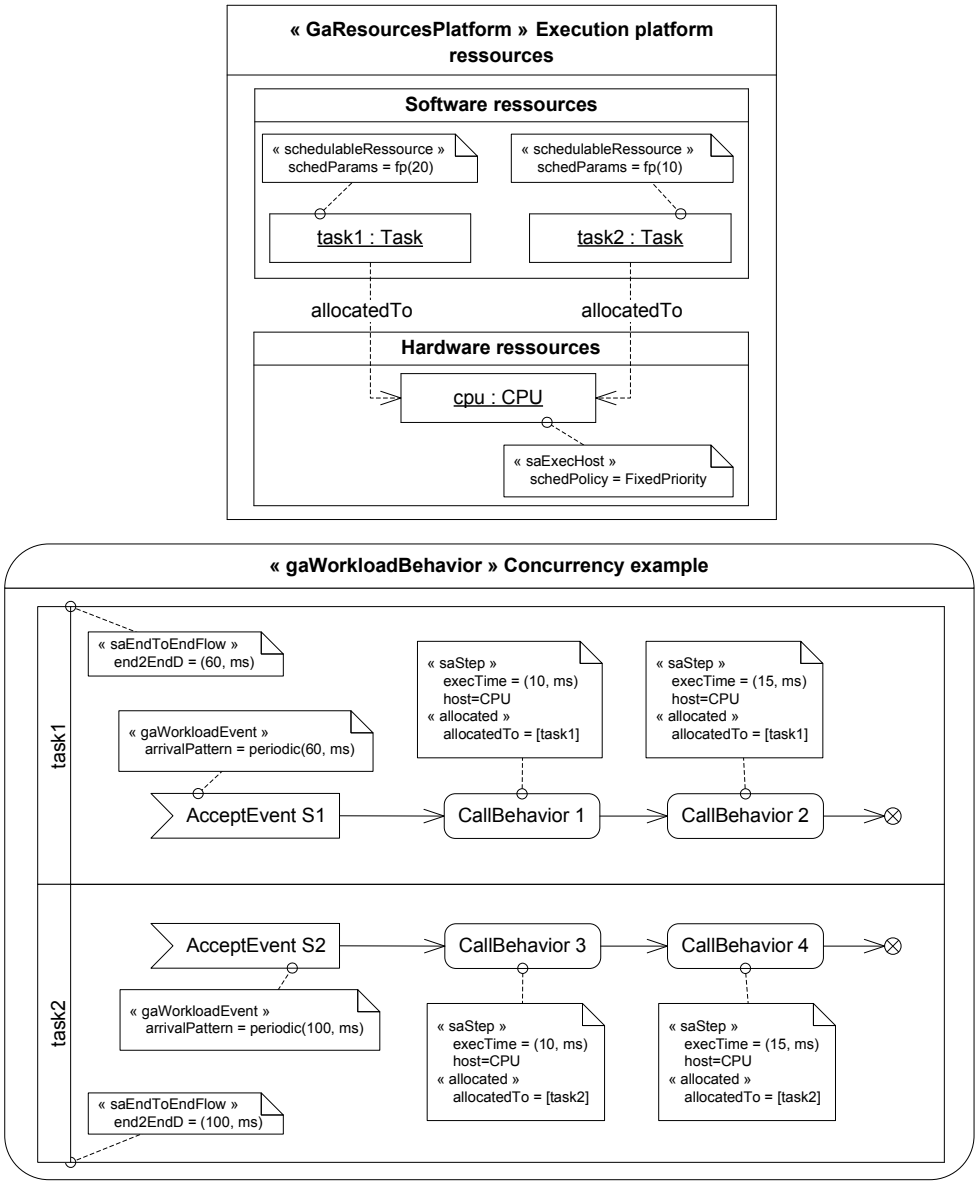


FIGURE 4.17 – Le modèle obtenue en appliquant la méthodologie Optimum

La partie supérieure de la figure correspond à la plateforme d'exécution. Elle contient une seule unité physique de calcul et deux ressources d'exécution task1 et task2. L'algorithme d'ordonnancement que nous avons choisi est un ordonnancement à priorité fixe de type taux monotone (Rate-monotonic) [94]. La tâche 1 est plus prioritaire que la tâche 2. La partie

inférieure de la figure correspond au modèle de comportement. Il contient deux scénarios composés de *CallBehaviorAction*. Le mapping que nous avons utilisé est un mapping par scénario. Donc toutes les fonctions d'un scénario sont exécutées par la même tâche. Quant aux événements déclencheurs, ils sont modélisés par des signaux. Leur politique d'arrivée est périodique.

Le modèle d'instance des classes du modèle d'exécution qui correspond au diagramme d'activité de notre exemple est présenté par la figure 4.18.

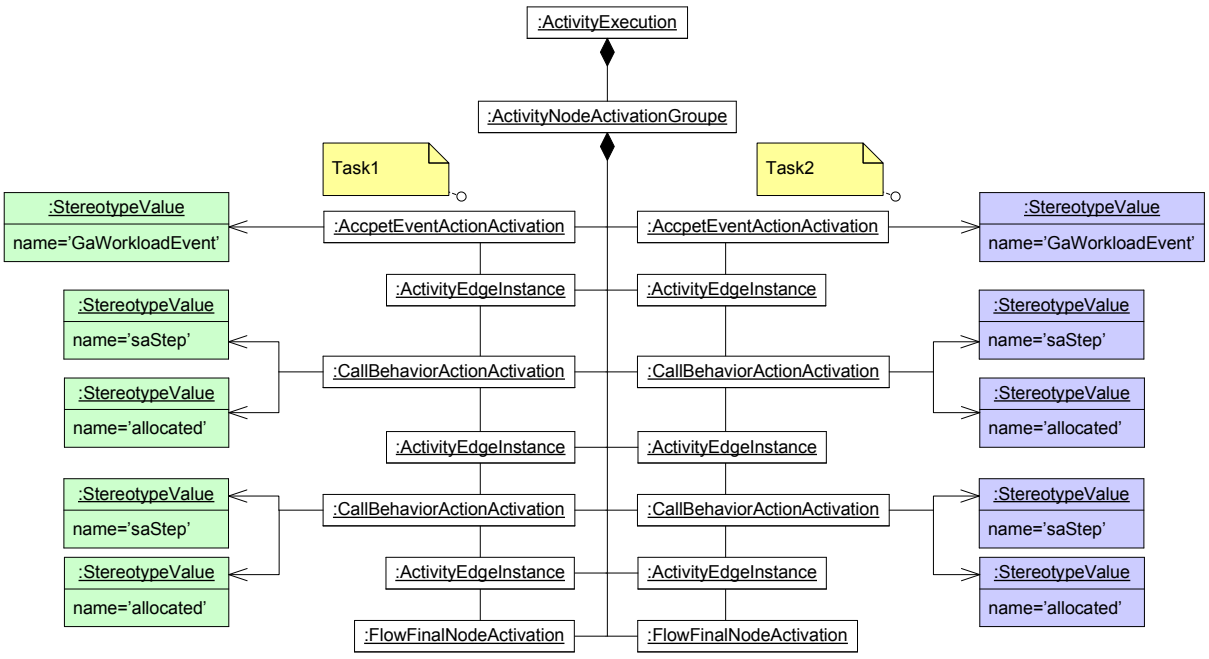


FIGURE 4.18 – Modèle d'instance du modèle d'exécution du diagramme d'activité du cas d'étude

4.3 La sémantique d'exécution d'Optimum

La sémantique d'exécution associée au modèle obtenu par Optimum varie en fonction de différents paramètres notamment, le nombre d'unité physique (exécution mono processeur ou multiprocesseur), l'algorithme d'ordonnancement (priorité, préemption). Pour notre cas d'étude, la sémantique associée est la suivante :

- ≻ Chaque scénario commence à la réception du signal déclencheur.
- ≻ Une fois le scénario déclenché, toutes ses actions sont exécutées jusqu'à la fin du flot (FinalFlowNode).
- ≻ L'algorithme d'ordonnancement est de type priorité fixe à taux monotone.
- ≻ L'exécution d'un scénario est atomique, donc il n'y a pas de préemption.
- ≻ Quand deux événements (signaux) arrivent en même temps, le scénario à exécuter est sélectionné selon la priorité assignée à la tâche sur laquelle il est alloué.

➢ Toutes les exécutions doivent respecter les contraintes de temps spécifiées dans le modèle.

Afin de simuler le modèle par le moteur d'exécution de fUML, on doit ajouter cette sémantique dans le modèle d'exécution en respectant les extensions que nous avons proposées. Pour cela nous avons introduit les classes suivantes :

➢ La classe **RMSelectNextActionStrategy** pour définir la politique d'ordonnancement que nous avons adopté. L'algorithme d'ordonnancement à taux monotone est implémenté dans l'opération *selectionNextAction* de cette classe.

➢ La classe **NFPEvaluateTaggedValueStrategy** pour définir le mécanisme d'évaluation des propriétés exprimées en VSL des stéréotypes NFPDuration . Le mécanisme d'évaluation est implémenté dans l'opération *evaluateTaggedValue*. Pour une expression du type *schedParams =[fp(30)]* l'opération d'évaluation renvoie la valeur 30 qui sera utilisé dans la politique d'ordonnancement.

La figure 4.19 expose les classes ajoutées dans le modèle d'exécution. L'implémentation de ces classes est présentée dans l'annexe A.

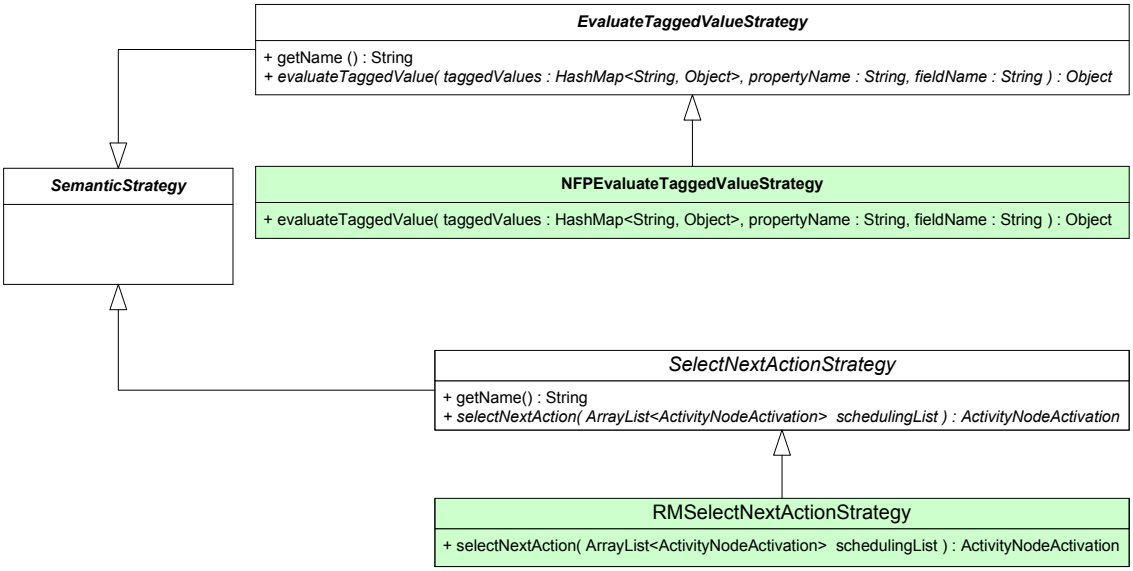


FIGURE 4.19 – Les classes du modèle d'exécution de fUML pour définir la sémantique d'exécution d'Optimum

Afin de générer des traces d'exécution qui permettent de vérifier et valider efficacement le comportement à simuler, nous avons utilisé la librairie OTF [95] (Open Trace Format). Cette librairie permet de générer des traces d'exécution stockées dans un fichier. Ces traces permettent de visualiser les différents flots d'exécution ainsi que les propriétés temporelles de l'exécution. Pour visualiser un fichier de trace, il faut utiliser des outils destinés à la visualisation des traces OTF tels que Vampir [96] et le plugin G-Eclipse [97]. Dans notre cas, nous avons choisi Vampir car il offre une représentation intuitive et détaillée.

La librairie d'OTF fournit une API libre destinée à la lecture et l'écriture des traces d'exécution volumineuses qui peuvent atteindre 1 gigaoctet de données. Cette API est utilisée essentiellement dans le domaine du calcul à haute performance, pour analyser les exécutions et les communications d'un nombre important de processeurs. L'idée principale d'OTF est de stocker les informations des événements de l'exécution (occurrence d'un événement, temps de début et de fin d'exécution d'une fonction, etc.) dans des enregistrements d'événements (fichier) qui seront triés par le temps. Dans cette étude, nous utilisons l'API OTF pour enrichir le code de la politique d'ordonnancement (l'opération *selectNextAction*) afin de générer l'information que nous voulons observer. Le document [98] donne une spécification complète de toute l'API d'OTF.

La figure 4.20 présente la trace d'exécution obtenue par la simulation du cas d'étude. La simulation a été réalisé sur un intervalle de temps de 220 ms.

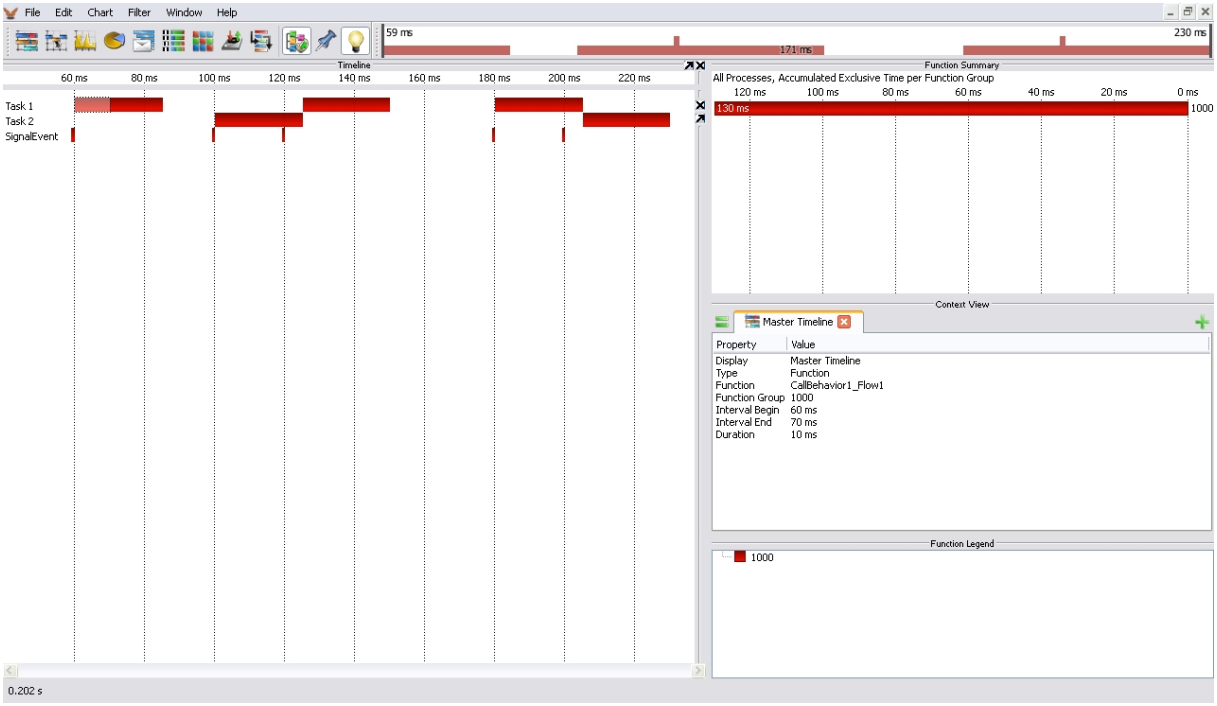


FIGURE 4.20 – Visualisation de la trace d'exécution OTF de la simulation du cas d'étude par l'outil Vampir

On remarque que nous avons deux flots d'exécution concurrents task1 et task2. Nous avons ajouté le troisième flot SignalEvent dans la trace pour permettre de connaitre la date d'occurrence de l'événement déclencheur des flots d'exécutions. Les rectangles rouges représentent les exécutions des différents flots. Les informations temporelles sur l'exécution d'un flot sont obtenues en cliquant sur le rectangle correspondant. La maniéré dont nous avons utilisé l'API OTF pour généré cette trace d'exécution est expliqué dans l'annexe A. La trace d'exécution que nous avons présentée illustre un cas idéal d'exécution qui ne contient aucun problème à détecter. Le but derrière ce type de trace est de pouvoir détecter une erreur et la corriger le plus tôt possible dans le flot de conception d'une application. Pour cela, nous

allons montrer deux cas où nous détecterons un dépassement d'échéance.

Dans le premier cas, nous modifions le temps d'exécution de *CallBehavior1* et de *CallBehavior2* qui s'exécutent sur la tâche 1. Nous affectons 30 ms à *CallBehavior1* et 35 ms à *CallBehavior2*. La figure 4.21 montre la trace d'exécution obtenue par le moteur d'exécution de fUML. On remarque un premier dépassement d'échéance pour *CallBehavior1* à t0+120 ms et un deuxième dépassement d'échéance t0+180 ms pour *CallBehavior2*. Le moteur d'exécution de fUML affiche un message d'erreur à chaque échéance dépassée.

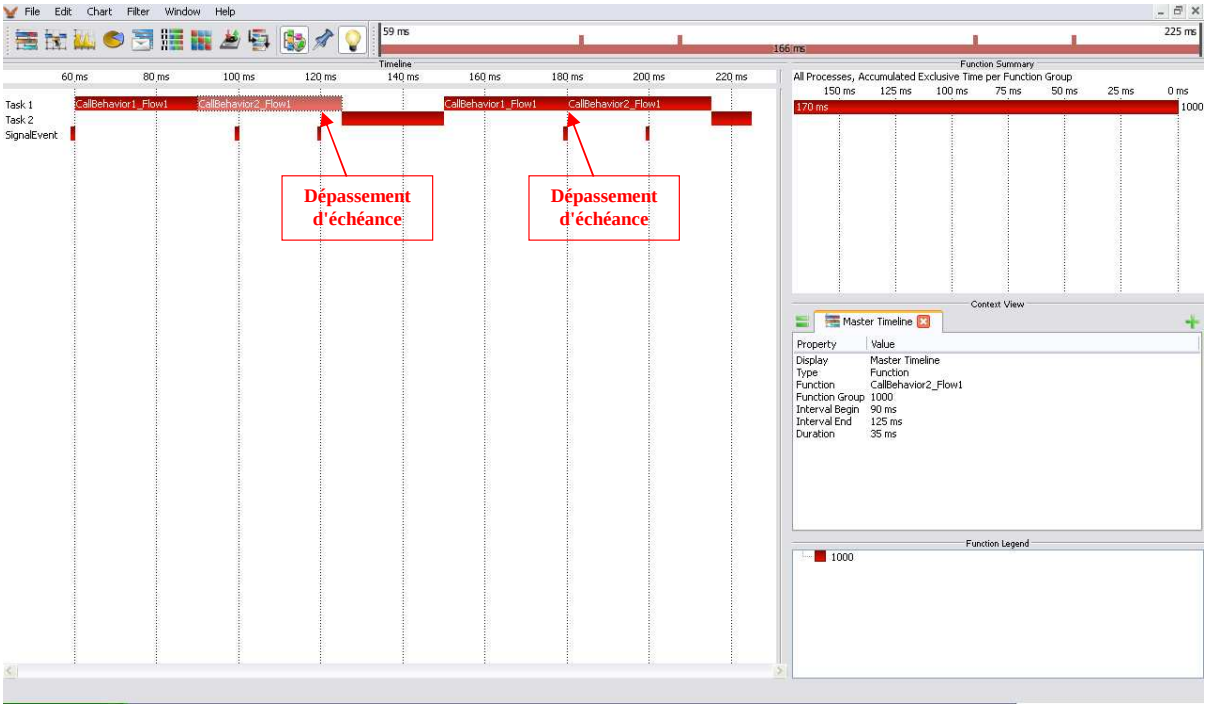


FIGURE 4.21 – Visualisation de la trace d'exécution OTF de la simulation du cas d'étude (Dépassement d'échéance 1)

Dans le deuxième cas, nous modifions les temps d'exécution de *CallBehavior3* et *CallBehavior4* qui s'exécutent sur la tâche 2. Nous affectons 30 ms à *CallBehavior3* et 30 ms à *CallBehavior4*. La figure 4.22 montre la trace d'exécution obtenue par le moteur d'exécution de fUML. On remarque bien un dépassement d'échéance pour *CallBehavior2* à t0+180 ms.

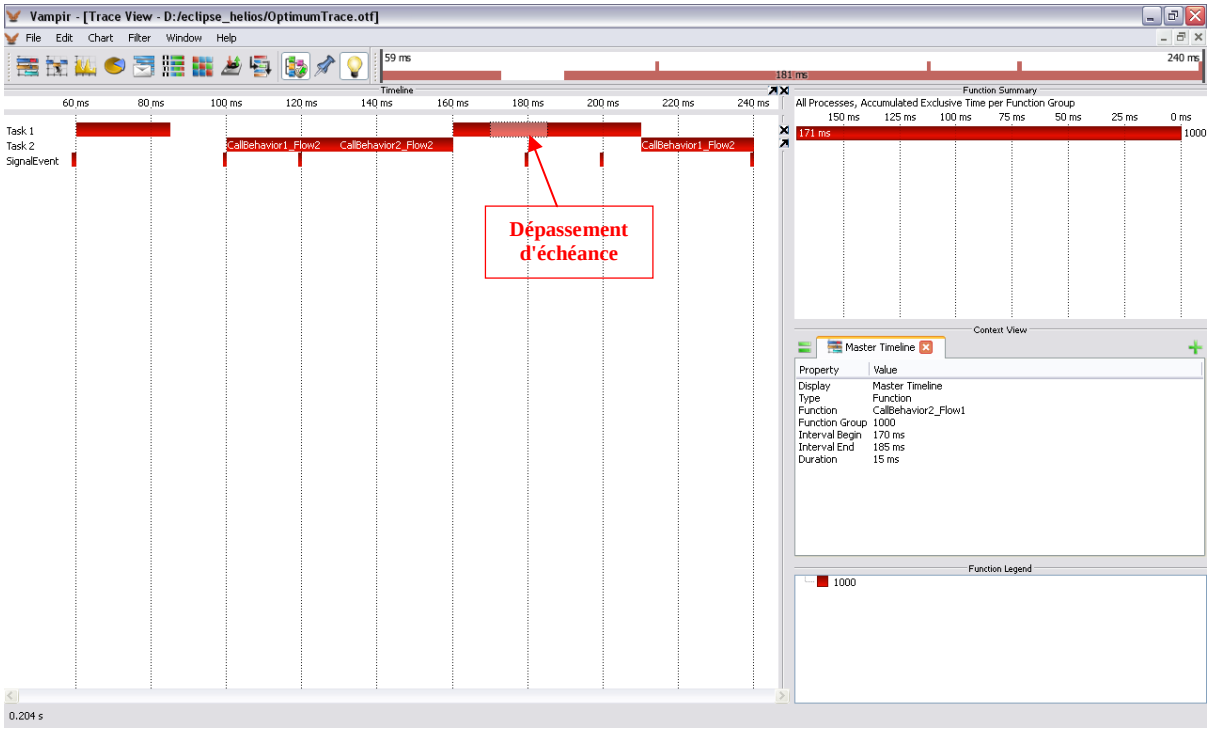


FIGURE 4.22 – Visualisation de la trace d'exécution OTF de la simulation du cas d'étude (Dépassement d'échéance 2)

CHAPITRE 5

Conclusion

1 Résumé

Les travaux présentés dans ce manuscrit de thèse traitent de l'exécution et de la simulation pour le domaine des systèmes temps réel embarqués. Ce travail s'inscrit dans le contexte de l'ingénierie dirigée par les modèles. L'objectif de ces travaux était de mettre en øeuvre un moteur d'exécution de modèles exploitant les hypothèses sur la sémantique d'exécution des modèles à un niveau d'abstraction élevé afin de pouvoir simuler ces modèles le plus tôt possible dans le flot de conception d'une application.

Pour cela, nous avons exploité le standard OMG «Semantics of a Foundational Subset for Executable UML Models» (fUML). Ce standard propose une formalisation de la sémantique d'exécution pour un sous-ensemble du méta-modèle d'UML. La sémantique est spécifiée dans un style opérationnel, c'est le type de description le plus adapté pour décrire les aspects comportementaux. Cette description sémantique est organisée sous forme d'un modèle d'exécution suffisamment détaillé pour permettre l'interprétation des modèles. L'utilisateur peut modéliser la structure et le comportement de son application puis exécuter son modèle en utilisant cet interpréteur pour valider et vérifier ses choix de conception dans un niveau d'abstraction élevé.

Premièrement, nous avons présenté le contexte général de ces travaux. Nous avons survolé les concepts essentiels de l'ingénierie dirigée par les modèles ainsi que les aspects importants des systèmes temps réel embarqués. Nous avons mené ensuite un état de l'art sur les différentes techniques de description de la sémantique des langages de modélisation et les approches et outils permettant la définition de la sémantique et la simulation des modèles. Par la suite, nous avons présenté en détail le standard fUML en identifiant ses limites vis-à-vis des besoins de modélisation et de sémantique d'exécution des modèles de systèmes temps réel. Dans la dernière partie, nous avons présenté notre approche pour l'exploitation de la sémantique d'exécution pour la simulation des modèles temps réel. Cette approche se base sur le modèle d'exécution de fUML. Elle étend le modèle d'exécution de fUML afin de prendre en compte les aspects relevant du domaine du temps réel. Les différentes réalisations que nous avons effectuées sont :

- Nous avons introduit un ordonnanceur. Cela permet de contrôler les exécutions des actions et donne le moyen de définir plusieurs politiques d'ordonnancement.
- Nous avons introduit une horloge. Cela permet de prendre en compte la notion du temps dans les exécutions des modèles.
- Nous avons ajouté les mécanismes nécessaires pour prendre en compte l'application des profils sur les modèles. Cela permet de paramétrer le modèle d'exécution de fUML afin d'interpréter une sémantique d'exécution particulière impliquée par l'application d'un profil.
- Nous avons montré par un cas d'étude comment utiliser ces extensions pour paramétrer le modèle d'exécution de fUML en fonction d'une sémantique particulière.

La sémantique que nous avons abordée est celle d'un sous-profil de MARTE appliqué à l'analyse d'ordonnancement temps réel.

≻ Nous avons introduit le format de traces OTF. Cela permet de vérifier et valider efficacement les résultats de simulation des modèles.

≻ Toutes ces extensions sont implémentées sous forme d'un plugin intégré à l'outil Papyrus.

La solution proposée répond aux quatre critères que nous avons définis dans l'introduction, en résumé :

≻ Les différentes extensions réalisées au niveau du modèle d'exécution de fUML, en particulier l'ordonnanceur et l'horloge permettent de simuler des comportements temporisés et concurrents.

≻ La solution proposée s'appuie sur la sémantique précise du standard fUML. De plus, la génération de traces de simulation dans un format standardisé permet l'exploitation de ces informations pour une activité d'analyse.

≻ L'extension qui gère l'application des profils sur les modèles et le paramétrage du modèle d'exécution de fUML, offre une grande flexibilité permettant de capturer les particularités de différents domaines.

≻ Le standard fUML respect par défaut les préconisations du MDA, cela à faciliter considérablement l'intégration du moteur d'exécution obtenue dans un flot de conception IDM. Cela est concrétisé par un plugin de simulation de modèles dans l'outil Papyrus.

2 Perspectives

Les perspectives des travaux présentés dans ce mémoire suivent différents axes de réflexion et se situent dans des contextes de recherche différents.

A court terme nous devons essentiellement définir une méthodologie générique qui donne des règles méthodologiques de paramétrage du modèle d'exécution en fonction de la sémantique d'un profil.

Pour la partie implémentation, nous devons améliorer l'aspect graphique des simulations. Nous devons ajouter l'aspect animation de modèles afin de pouvoir visualiser l'évaluation des modèles dans le temps. Un autre aspect d'implémentation que nous devons améliorer est celui de la visualisation de traces d'exécution. Pour éviter la dépendance à un outil externe de visualisation tel que Vampir, nous devons développer notre propre plugin de visualisation de traces dans Papyrus.

A moyen terme, nous prévoyant d'enrichir le domaine de simulation de modèle UML pour le temps réel. Notre approche sera raffinée de telle sorte que les extensions proposées

s'appuient plus explicitement sur des fondements théoriques de la simulation à événement discret. Ce raffinement devrait permettre d'affiner les résultats de simulation, de faire de l'exploration d'architectures en prenant en compte des modèles de plateformes d'exécution très abstraits

A long terme, nous devons réfléchir à une généralisation du principe d'extraction du contrôle du modèle d'exécution, de façons à permettre à un outil extérieur de piloter l'exécution et d'interagir avec le moteur d'exécution. Il serait intéressant aussi de réfléchir à l'aspect des exécutions distribuées et de voir comment l'intégrer au modèle d'exécution de fUML.

ANNEXE A

Implémentation des algorithmes de la sémantique du cas d'étude

1 Implémentation de la classe *RMSelectNextActionStrategy*

Cette classe implémente en Java la sémantique d'exécution des modèles obtenues par l'application de la méthodologie Optimum.

Listing A.1 – Class RMSelectNextAction

```
package fUML.Semantics.Loci;

/**
 *
 * RMSelectNextActionStrategy
 *
 * @author Abderraouf Benyahia { abderraouf.benyahia@supelec.fr }
 *
 */


public class RMSelectNextActionStrategy extends SelectNextActionStrategy {

   // Attributs pour capturer la sémantique d'Optimum

   private int                          t              = 1;
   private int                          state          = 0;
   private List<ActivityNodeActivation>       acceptActionList = new List<
       ActivityNodeActivation>();
   private List<ActivityNodeActivation>       cba              = new List<
       ActivityNodeActivation>();
   private int                          maxAccept         = 10;
```

```
private int[]                       lastAcceptStart      = new int[maxAccept];
private int                         periodReadyAccept;
private enum                        state { 0,1,2,3,4 };


// Attributs pour la gestion des traces OTF

private OTF                         otf                  = OTF.INSTANCE;
private OTF.OTF_Writer                 writer;
private OTF.OTF_FileManager               manager;
private int                         randomFuncID         = 0;
private int                         randomFuncIDSignal   = 500;

/*=================================================================*/
 /* Méthode de sélection d'une action de la liste en fonction de la
    sémantique d'Optimum */
/*=================================================================*/
@Override
public ActivityNodeActivation selectNextAction(List<ActivityNodeActivation>
    schedulingList, Locus locus) {

   switch (state) {
      case 0: {

         // Initialisation du file manager d'OTF pour l'enregistrement des
            traces
         manager = otf.OTF_FileManager_open(1);
         assert (manager != null);
         writer = otf.OTF_Writer_open("OptimumTrace", 1, manager);
         assert (writer != null);

         // Création des enregistrements des traces pour chaque flot d'
            exécution
         otf.OTF_Writer_writeDefTimerResolution(writer, 0, 1000);
         otf.OTF_Writer_writeDefProcess(writer, 0, 1, "Task 1", 0);
         otf.OTF_Writer_writeDefProcess(writer, 0, 2, "Task 2", 0);
         otf.OTF_Writer_writeDefProcess(writer, 0, 3, "SignalEvent", 0);
         otf.OTF_Writer_writeDefProcessGroup(writer, 0, 3, "GaWorkloadBehavior
            ");

         // Lancement de la capture des traces d'execution pour chaque flot d'
            exécution
         otf.OTF_Writer_writeBeginProcess(writer, 0, 1);
         otf.OTF_Writer_writeBeginProcess(writer, 0, 2);
         otf.OTF_Writer_writeBeginProcess(writer, 0, 3);


         this.state = 1;
      }
```

```
case 1: {

   for (int i = 0; i < maxAccept; i++) {
      lastAcceptStart[i] = -1;
   }

   // Chercher les callBehaviorAction qui appellent une activité sur
       laquelle stéréotype SaEndtoEndFlow est appliqué
   for (int i = 0; i < schedulingList.size(); i++) {
      if (schedulingList.get(i) instanceof CallBehaviorActionActivation)
          {
         CallBehaviorAction callBehaviorAction = (CallBehaviorAction)
             schedulingList.get(i).node;
         Activity calledActivity = (Activity) callBehaviorAction.
             getBehavior();
         EList<Stereotype> appliedStereotypes = calledActivity.
             getAppliedStereotypes();
         if ((appliedStereotypes, "SaEndtoEndFlow") == 1) {
            return schedulingList.get(i);
         }
      }
   }

   this.state = 2;
}

case 2: {

   // Démarrer les accepetEventAction en fonction de la priorité de la
       tâche sur laquelle les actions sont allouées
   for (int i = 0; i < schedulingList.size(); i++) {
      if (schedulingList.get(i) instanceof AcceptEventActionActivation)
          {
         this.acceptActionList.add(schedulingList.get(i));
         return schedulingList.get(i);
      }
   }

   this.state = 3;

}

case 3: {

    // Algorithme de choix de des actions selon la politique RM
   while (locus.clock.getTime() < simulationCycles) {

      int indiceS = sendSignal();
```

```
            if (indiceS != -1)
               return acceptActionList.get(indiceS);

            if (!schedulingList.isEmpty()) {
               for (int i = 0; i < schedulingList.size(); i++) {
                  int isStereotyped = stereotypeIsApplied(schedulingList.get(i
                      ).node.getAppliedStereotypes(), "SaStep");

                  if ((isStereotyped == 0)) {
                     return schedulingList.get(i);

                  } else {
                     this.cba.add(0, schedulingList.get(i));
                     ordonerCBA_Nonpreamptif();
                     schedulingList.remove(i);
                  }
               }
            } else {
               ActivityNodeActivation nodeActivationLate = null;
               if (!cba.isEmpty()) {
                  for (int i = 0; i < cba.size(); i++) {
                     if (cba.get(i).isReady()) {
                        Debug.myPrintlnForSched("[Debug] start time of: " +
                            cba.get(i).node.getName() + " =" + locus.clock.
                            getTime());
                        this.randomFuncID++;

                        // Ecriture de la trace OTF
                        otf.OTF_Writer_writeDefFunction(writer, 0, this.
                            randomFuncID, cba.get(i).node.getName(), 1000, 0);
                        if (getTaskName(cba.get(i)).compareTo("task1") == 0) {
                           otf.OTF_Writer_writeEnter(writer, locus.clock.
                               getTime(), this.randomFuncID, 1, 0);
                        } else {
                           if (getTaskName(cba.get(i)).compareTo("task2") ==
                               0) {
                              otf.OTF_Writer_writeEnter(writer, locus.clock.
                                  getTime(), this.randomFuncID, 2, 0);
                           } else {
                              otf.OTF_Writer_writeEnter(writer, locus.clock.
                                  getTime(), this.randomFuncID, 4, 0);
                           }

                        }

                        ActivityNodeActivation nodeActivation = cba.get(i);
                        int execTime = getExecTime(nodeActivation);
                        nodeActivationLate = null;
                        nodeActivationLate = updateTime(execTime, locus);
```

```
                        if (nodeActivationLate != null) {
                            nodeActivationLate.receiveOffer();
                            schedulingList.add(nodeActivationLate.outgoingEdges
                                .get(0).target);
                        }

                        Debug.myPrintlnForSched("[Debug] end time of: " + cba.
                            get(i).node.getName() + " =" + locus.clock.getTime
                            ());

                        // Ecriture de la trace OTF
                        if (getTaskName(cba.get(i)).compareTo("task1") == 0) {
                            otf.OTF_Writer_writeLeave(writer, locus.clock.
                                getTime(), this.randomFuncID, 1, 0);
                        } else {
                            if (getTaskName(cba.get(i)).compareTo("task2") ==
                                0) {
                                otf.OTF_Writer_writeLeave(writer, locus.clock.
                                    getTime(), this.randomFuncID, 2, 0);
                            } else {
                                otf.OTF_Writer_writeLeave(writer, locus.clock.
                                    getTime(), this.randomFuncID, 4, 0);
                            }

                        }

                        cba.remove(i);
                        return nodeActivation;
                    }
                }
            } else { }

            locus.clock.advanceTime(1);

        }
    }

}
case 4: {

    // Fermeture du fichier de traces OTF et fin de la simulation
    otf.OTF_Writer_writeEndProcess(writer, locus.clock.getTime(), 1);
    otf.OTF_Writer_writeEndProcess(writer, locus.clock.getTime(), 2);
    otf.OTF_Writer_writeEndProcess(writer, locus.clock.getTime(), 3);
    otf.OTF_Writer_close(writer);
    otf.OTF_FileManager_close(manager);

}
}
```

```
    return null;
}

/*================================================================*/
/* Méthode de mise à jour du temps                                */
/*================================================================*/

ActivityNodeActivation updateTime(int advanceTime, Locus locus) {
    if (advanceTime == 1) {
        locus.clock.advanceTime(1);
    } else {
        for (int i = 1; i <= advanceTime; i++) {
            locus.clock.advanceTime(1);
            int indiceS = sendSignal();
            if (indiceS != -1) {
                locus.clock.advanceTime(advanceTime - i);
                return acceptActionList.get(indiceS);
            }
        }
    }
    return null;
}

/*================================================================*/
/* Méthode d'ordonnancement des CallBehaviorAcrion pour la version préemptive
    de RM */
/*================================================================*/

void ordonerCBA() {
    int maxPriorityJ = 0;
    if (this.cba.size() > 1) {
        for (int i = this.cba.size() - 1; i >= 0; i--) {
            for (int j = 0; j <= i; j++) {
                if (getPriority(this.cba.get(maxPriorityJ)) < getPriority(this.cba
                    .get(j))) {
                    maxPriorityJ = j;
                }
                if (maxPriorityJ != i) {
                    ActivityNodeActivation tempNode = this.cba.get(maxPriorityJ);
                    this.cba.add(maxPriorityJ, this.cba.get(i));
                    this.cba.remove(maxPriorityJ + 1);
                    this.cba.add(i, tempNode);
                    this.cba.remove(i + 1);
                }
            }
        }
    }
}
```

```
/*=====================================================================*/
/* Méthode d'ordonnancement des CallBehaviorAcrion pour la version non
    préemptive de RM */
/*=====================================================================*/

void ordonerCBA_Nonpreamptif() {
   int maxPriorityJ = 0;
   if (this.cba.size() > 1) {
      for (int i = this.cba.size() - 1; i > 0; i--) {
         for (int j = 1; j <= i; j++) {
            if (getPriority(this.cba.get(maxPriorityJ)) < getPriority(this.cba
                .get(j))) {
               maxPriorityJ = j;
            }
            if (maxPriorityJ != i) {
               ActivityNodeActivation tempNode = this.cba.get(maxPriorityJ);
               this.cba.add(maxPriorityJ, this.cba.get(i));
               this.cba.remove(maxPriorityJ + 1);
               this.cba.add(i, tempNode);
               this.cba.remove(i + 1);
            }
         }
      }
   }
}

/*=====================================================================*/
/* Méthode pour récupérer le temps d'exécution des actions             */
/*=====================================================================*/

int getExecTime(ActivityNodeActivation node) {
   StereoTypeValue saStep = node.getExecutionLocus().stereoTypeFactory.
       findStereoTypeValue("SaStep", node.stereoTypeValues);

   Object o = node.getExecutionLocus().evaluator.evaluateTaggedValueV1(saStep.
       getTaggedValues(), "execTime");
   EList<String> oo = (EList<String>) o;

   int execTime = new Integer(oo.get(0));
   return execTime;
}

/*=====================================================================*/
/* Méthode pour récupérer la priorité des actions                      */
/*=====================================================================*/

int getPriority(ActivityNodeActivation node) {
   StereoTypeValue allocated = node.getExecutionLocus().stereoTypeFactory.
       findStereoTypeValue("Allocated", node.stereoTypeValues);
```

```
    Object o = node.getExecutionLocus().evaluator.evaluateTaggedValueV1(
        allocated.getTaggedValues(), "allocatedTo");
    BasicEObjectImpl tmp = ((BasicInternalEList<BasicEObjectImpl>) o).get(0);

    Property task = (Property) UMLUtil.getBaseElement(tmp);
    String taskName = task.getName();

    EList<Stereotype> st = task.getAppliedStereotypes();
    EList<String> oo = (EList<String>) task.getValue(stereotypeIsAppliedV2(st,
        "SchedulableResource"), "schedParams");

    int prio = new Integer(oo.get(0));
    return prio;
}

/*================================================================*/
/* Méthode pour récupérer le nom de la tâche sur laquelle une action est
    exéctuée */
/*================================================================*/

String getTaskName(ActivityNodeActivation node) {
    StereoTypeValue allocated = node.getExecutionLocus().stereoTypeFactory.
        findStereoTypeValue("Allocated", node.stereoTypeValues);

    Object o = node.getExecutionLocus().evaluator.evaluateTaggedValueV1(
        allocated.getTaggedValues(), "allocatedTo");
    BasicEObjectImpl tmp = ((BasicInternalEList<BasicEObjectImpl>) o).get(0);

    Property task = (Property) UMLUtil.getBaseElement(tmp);
    String taskName = task.getName();

    return taskName;
}

/*================================================================*/
/* Méthode pour la génération d'évenements par envoie de signal    */
/*================================================================*/

int sendSignal() {
    for (int i = 0; i < acceptActionList.size(); i++) {

        ActivityNodeActivation aa = acceptActionList.get(i);
        List<StereoTypeValue> stv = aa.stereoTypeValues;
        Locus ll = aa.getExecutionLocus();
        StereoTypeValue ss = ll.stereoTypeFactory.findStereoTypeValue("
            GaWorkloadEvent", acceptActionList.get(i).stereoTypeValues);

        StereoTypeValue gaWorkloadEventAccepti = acceptActionList.get(i).
```

```
            getExecutionLocus().stereoTypeFactory.findStereoTypeValue("
            GaWorkloadEvent", acceptActionList.get(i).stereoTypeValues);

        ActivityNodeActivation a = acceptActionList.get(i);
        Locus l = a.getExecutionLocus();
        Object o = l.evaluator.evaluateTaggedValueV1(gaWorkloadEventAccepti.
            getTaggedValues(), "pattern");

        int periodAccepti = new Integer((String) acceptActionList.get(i).
            getExecutionLocus().evaluator.evaluateTaggedValueV1(
            gaWorkloadEventAccepti.getTaggedValues(), "pattern"));

        if ((acceptActionList.get(i).getExecutionLocus().clock.getTime() - (
            lastAcceptStart[i] + 1) >= periodAccepti) && (lastAcceptStart[i] ==
            -1)) {

            if ((acceptActionList.get(i).getExecutionLocus().clock.getTime() - (
                lastAcceptStart[i] + 1)) > periodAccepti) {
               Debug.myPrintlnForSchedErr("period violation !");
            }
            Debug.myPrintlnForSched("[Debug] Signale occurence time : " +
                acceptActionList.get(i).getExecutionLocus().clock.getTime() + " of
                : " + acceptActionList.get(i).node.getName());
            this.randomFuncIDSignal++;

            // Ecriture d'une trace OTF
            otf.OTF_Writer_writeDefFunction(writer, 0, this.randomFuncIDSignal,
                acceptActionList.get(i).node.getName(), 1000, 0);
            otf.OTF_Writer_writeEnter(writer, acceptActionList.get(i).
                getExecutionLocus().clock.getTime() - 1, this.randomFuncIDSignal,
                3, 0);
            otf.OTF_Writer_writeLeave(writer, acceptActionList.get(i).
                getExecutionLocus().clock.getTime(), this.randomFuncIDSignal, 3,
                0);

            lastAcceptStart[i] = acceptActionList.get(i).getExecutionLocus().
                clock.getTime();
            acceptActionList.get(i).receiveOffer();
            return -1;
        } else
            if (acceptActionList.get(i).getExecutionLocus().clock.getTime() -
                lastAcceptStart[i] >= periodAccepti && (lastAcceptStart[i] != -1))
                 {
               if (acceptActionList.get(i).getExecutionLocus().clock.getTime() -
                   lastAcceptStart[i] > periodAccepti) {
                  Debug.myPrintlnForSchedErr("period violation !");
               }
               Debug.myPrintlnForSched("[Debug] Signale occurence time : " +
                   acceptActionList.get(i).getExecutionLocus().clock.getTime() + "
```

```
                    of: " + acceptActionList.get(i).node.getName());
                this.randomFuncIDSignal++;

                // Ecriture d'une trace OTF
                otf.OTF_Writer_writeDefFunction(writer, 0, this.randomFuncIDSignal
                    , acceptActionList.get(i).node.getName(), 1000, 0);
                otf.OTF_Writer_writeEnter(writer, acceptActionList.get(i).
                    getExecutionLocus().clock.getTime() - 1, this.
                    randomFuncIDSignal, 3, 0);
                otf.OTF_Writer_writeLeave(writer, acceptActionList.get(i).
                    getExecutionLocus().clock.getTime(), this.randomFuncIDSignal,
                    3, 0);

            lastAcceptStart[i] = acceptActionList.get(i).getExecutionLocus().
                clock.getTime();
            return i;
          }

      }
      return -1;
   }
}
```

2 Implémentation de la classe *VSLEvaluateTaggedValueStrategy*

Cette classe implémente en Java le mécanisme d'évaluation des propriétés de MARTE exprimées en VSL.

Listing A.2 – Class VSLEvaluateTaggedValueStrategy

```
package fUML.Semantics.Loci;

import java.util.HashMap;

import org.eclipse.uml2.uml.ActivityNode;

/**
 *
 * VSLEvaluateTaggedValueStrategy
 *
 * @author Abderraouf Benyahia { abderraouf.benyahia@supelec.fr }
 *
 */

public class VSLEvaluateTaggedValueStrategy extends EvaluateTaggedValueStrategy {

   public Locus locus = null;
```

```
    /*================================================================*/
     /* Méthode de d'évaluation des propriétés de typde VSL */
    /*================================================================*/

    @Override
    public Object evaluateTaggedValue(HashMap<String, String> taggedValues, String
         propertyName, String fieldName) {

        String property = taggedValues.get(propertyName);
        String[] temp1 = property.split("\\(");
        String[] temp2 = temp1[1].split("\\)");
        String[] temp3 = temp2[0].split(" , ");

        HashMap<String, Object> filedValues = new HashMap<String, Object>();

        for (int i = 0; i < temp3.length; i++) {
            String[] temp4 = temp3[i].split("=");
            if (temp4.length > 1) {
                filedValues.put(temp4[0], temp4[1]);
            }
        }
        return filedValues.get(fieldName);
    }
}
```

Table des figures

Bibliographie

[1] Thierry Le Sergent, Yann Tanguy Alain Leguennec (Esterel Technologies), François Terrier, and Sébastien Gérard (CEA). Scade system, a comprehensive toolset for smooth transition from model-based system engineering to certified embedded control and display software. *White paper*.

[2] Best of show "embeddy" à embedded world 2011. http://www.esterel-technologies.com/news-events/press-releases/2011/Esterel-Technologies-Wins-Best-of-Show-at-Embedded-World-2011.

[3] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE – The European Journal for the Informatics Professional*, 5(2) :21–24, 2004.

[4] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2) :171–188, 2005.

[5] Colin Atkinson and Thomas Kühne. Model-driven development : A metamodeling foundation. *IEEE Softw.*, 20 :36–41, September 2003.

[6] E. Seidewitz. What models mean. *Software, IEEE*, 20(5) :26 – 32, sept.-oct. 2003.

[7] Jean marie Favre. Towards a basic theory to model model driven engineering. In *In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*, 2004.

[8] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *ASE*, pages 273–280, 2001.

[9] B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5) :19 – 25, sept.-oct. 2003.

[10] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152 :125–142, March 2006.

[11] Jean Bézivin. La transformation de mod(é)les. In *Ecole d'Eté d'Informatique CEA EDF INRIA, cours nÂ°6*, page 16, 2003.

[12] W3C. extensible stylesheet language transformation. http://www.w3.org/TR/xslt.

[13] W3C. An xml query language. http://www.w3.org/TR/xquery/.

[14] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *MoDELS Satellite Events*, pages 128–138, 2005.

[15] Thomas Stahl and Markus Völter. *Model Driven Software Development : Technology, Engineering, Management*. John Wiley & Sons, 2006.

[16] Etienne Juliot et Stéfane Lacrampe Jonathan Musset. *Acceleo 2.6 : Guide utilisateur*. Obeo, April 2008.

[17] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[18] Object Management Group, Inc. UML 2.3 Superstructure. Technical Report formal/2010-05-05, OMG.

[19] Object Management Group, Inc. Meta Object Facility (MOF) Core Specification Version 2.4.1. Technical Report formal/2011-08-07, OMG, 2011.

[20] Gul Agha. *Actors : a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[21] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[22] Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1. Technical Report formal/2011-01-01, OMG, 2011.

[23] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91 :145–164, January 2003.

[24] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[25] Object Management Group, Inc. Object Constraint Language Specification, version 2.0. Technical Report formal/2005-05-01, OMG, 2005.

[26] Eclipse. The graphical modeling framework (gmf). http://www.eclipse.org/gmf.

[27] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs : : a dsl for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 249–254, New York, NY, USA, 2006. ACM.

[28] David Harel and Bernhard Rumpe. Meaningful modeling : What's the semantics of "semantics" ? *Computer*, 37 :64–72, October 2004.

[29] James Bruck and Kenn Hussey. Which technique is right for you ? Technical report, International Business Machines, 2007.

[30] Object Management Group, Inc. UML Profile for CORBA and CORBAComponents Specification (CCCMP 1.0). Technical Report formal/2009-04-07, OMG, 2008.

[31] Object Management Group, Inc. UML Profile for System on a Chip (SoC). Technical Report formal/2006-08-01, OMG, 2006.

[32] Object Management Group, Inc. UML Profile For Schedulability, Performance, And Time, Version 1.1. Technical Report formal/2005-01-02, OMG, 2005.

[33] Object Management Group, Inc. UML Testing Profile (UTP) Version 1.1. Technical Report formal/2012-04-01, OMG, 2012.

[34] Object Management Group, Inc. UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems, V 1.1. Technical Report formal/2011-06-02, OMG, 2011.

[35] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, 1st edition, 2000.

[36] John A. Stankovic. *Real-Time Computing*. Departement of Computer Science, University of Massachusetts, Amherst, MA 01003, April 1992.

[37] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Trans. Softw. Eng.*, 19 :70–84, January 1993.

[38] Pascal Chevochot and Isabelle Puaut. An approach for fault-tolerance in hard real-time distributed systems. In *SRDS*, pages 292–293, 1999.

[39] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101, London, UK, 1991. Springer-Verlag.

[40] Inc Wind River Systems. Vxworks, publisher's website. http://www.windriver.com/products/vxworks/.

[41] M. Barabanov. A linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[42] OSEK. *Operating System Specification 2.2.3*. OSEK/VDX, February 2005.

[43] H. Chetto and J Delacroix. Minimisation des temps de réponse des tâches sporadiques en présence des tâches périodiques. *Real-Time Systems*, 5 :39–52, March 1993.

[44] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. In John A. Stankovic and K. Ramamritham, editors, *Tutorial : hard real-time systems*, pages 174–189. IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[45] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-software co-design of embedded systems : the POLIS approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[46] Bran Selic. Using uml for modeling complex real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 250–260, London, UK, 1998. Springer-Verlag.

[47] Bruce Powel Douglass. *Doing hard time : developing real-time systems with UML, objects, frameworks, and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[48] Gérard S, Terrier F, Dubois H, Mraidha C, and Baudry B. L'ingénierie des modéles pour les systémes temps-rèel. *As CNRS sur le mda(model driven architecture)*, pages 76–95, Avril 2003.

[49] Glynn Winskel. *The formal semantics of programming languages : an introduction*. MIT Press, Cambridge, MA, USA, 1993.

[50] Patrick Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 843–993. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[51] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12 :576–580, October 1969.

[52] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[53] Joseph E. Stoy. *Denotational semantics : the Scott-Strachey approach to programming language theory / by Joseph E. Stoy*. MIT Press, Cambridge, Mass. :, 1977.

[54] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9 :223–252, September 1977.

[55] Peter D. Mosses. The varieties of programming language semantics. In *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, TCS '00, pages 624–628, London, UK, 2000. Springer-Verlag.

[56] Benoit Combemale, Sylvain Rougemaille, Xavier Crégut, Frédéric Migeon, Marc Pantel, and Christine Maurel. Expériences pour décrire la sémantique en ingénierie des modèles. In Hermes Sciences/Lavoisier, editor, *2ième journées sur l'Ingénierie Dirigée par les Modèles (IDM, in french)*, pages 17–34, Lilles, France, June 2006.

[57] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. An action semantics for mof 2.0. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1304–1305, New York, NY, USA, 2006. ACM.

[58] Pierre alain Muller, Franck Fleurey, and Jean marc JÃ©zÃ©quel. Weaving executability into object-oriented meta-languages. In *in : International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713 (2005*, pages 264–278. Springer, 2005.

[59] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[60] Gabriele Taentzer. Agg : A graph transformation environment for modeling and validation of software. In John Pfaltz, Manfred Nagl, and Boris BÃ¶hlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin / Heidelberg, 2004.

[61] J. H. Hausmann. *Dynamic Meta Modeling. A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, Universität Paderborn, Germany, 2005.

[62] Tony Clark, Andy Evans, Paul Sammut, and James Willans. *Applied Metamodelling - A Foundation for Language Driven Development*. CETEVA, second edition, 2008.

[63] Tony Clark, Andy Evans, Paul Sammut, and James Willans. An executable metamodelling facility for domain specific language design. In *The 4th OOPSLA Workshop on Domain-Specific Modeling*, 2004.

[64] Didier Vojtisek. How to use and write a model checker with kermeta. In *Model checking manual*. Inria.

[65] Benoit Combemale, Xavier Crégut, Jean-Patrice Giacometti, Pierre Michel, and Marc Pantel. Introducing Simulation and Model Animation in the MDE Topcased Toolkit. In *4th European Congress EMBEDDED REAL TIME SOFTWARE (ERTS)*. SIA & SEE, January 2008.

[66] Eclipse. An open development platform. http://www.eclipse.org.

[67] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[68] Benoit Combemale, Xavier Crégut, Pierre Michel, Marc Pantel, and Sylvain Rougemaille. TOPCASED - Model Simulation : Adding model simulation to TOPCASED. Rapport de contrat D02b, IRIT, Université Paul Sabatier, Toulouse, février 2007.

[69] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 35–43, New York, NY, USA, 2005. ACM.

[70] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, 2001.

[71] Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors. *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.

[72] E. Berger and James K. Huggins. Abstract state machines 1988-1998 : Commented asm bibliography. *Bulletin of the EATCS*, 64, 1998.

[73] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. The graph rewriting and transformation language : Great. *Electronic Communications of the EASST*, 1, 2006.

[74] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, Martin von Löwis, and Andreas Prinz. Formal definition of sdl-2000 - compiling and running sdl specifications as asm models. 7(11) :1024–1049, nov 2001.

[75] Carlos Delgado Kloos. *Formal Semantics for VHDL*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[76] No Magic Inc. *MagicDraw, user guide, version 17.0.1*, 2011.

[77] No Magic Inc. *CAMEO SIMULATION TOOLKIT, user guide, version 17.0.1*, 2011.

[78] Object Management Group, Inc. OMG Systems Modeling Language (OMG SysML), v1.3. Technical Report formal/2012-06-02, OMG, 2012.

[79] No Magic Inc. *MagicDraw Open API, user guide, version 17.0.1*, 2011.

[80] W3C. State chart xml (scxml) : State machine notation for control abstraction w3c working draft 16 february 2012, 2012.

[81] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, June 1987.

[82] Abstract Solutions. Vxworks, publisher's simulator. http://www.kc.com/PRODUCTS/iuml/index.php.

[83] Ian Wilkie, Adrian King, Mike Clarke, and al. Uml asl reference guide. http://www.kc.com/download/index.php.

[84] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1) :127–144, January 2003.

[85] C Hardebolle. *Composition de modéles pour la modélisation multi-paradigme du comportement des systémes*. PhD thesis, Université Paris Sud - Paris XI, France, 2008.

[86] Object Management Group, Inc. Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0. Technical Report formal/2011-02-01, OMG, 2011.

[87] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[88] Modeldriven. Foundational uml reference implementation. http://portal.modeldriven.org/project/foundationalUML.

[89] Bran Selic. Using uml to model complex real-time architectures. In *OMER*, pages 16–21, 2001.

[90] Agnes Lanusse, Sébastien Gérard, and François Terrier. Real-time modeling with uml : The accord approach. In *UML*, pages 319–335, 1998.

[91] Sébastien Revol, Safouan Taha, FranÃ§ois Terrier, Alain Clouard, Sébastien Gérard, Ansgar Radermacher, and Jean-Luc Dekeyser. Unifying HW Analysis and SoC Design Flows by Bridging Two Key Standards : UML and IP-XACT. In *DIPES'2008 IFIP conference*, pages 69–78, 2008.

[92] OMG. Stereotype application issue. http://www.omg.org/issues/uml2-rtf.open.html#Issue15001.

[93] Chokri Mraidha, Sara Tucci-Piergiovanni, and Sebastien Gerard. Optimum : a marte-based methodology for schedulability analysis at early design stages. *SIGSOFT Softw. Eng. Notes*, 36 :1–8, January 2011.

[94] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[95] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In *International Conference on Computational Science (2)*, pages 526–533, 2006.

[96] Center for Information Services and High Performance Computing (ZIH). The vampir tool. http://www.vampir.eu/.

[97] H. Gjermundrod, M. D. Dikaiakos, M. Stumpert, P. Wolniewicz, and H. Kornmayer. g-eclipse - an integrated framework to access and maintain grid resources. In *Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, GRID '08, pages 57–64, Washington, DC, USA, 2008. IEEE Computer Society.

[98] Holger Brunst Andreas Knüpfer. Open trace format api specification version 1.1. Center for High Performance Computing University of Dresden, Germany, 2006.